
Hibernate

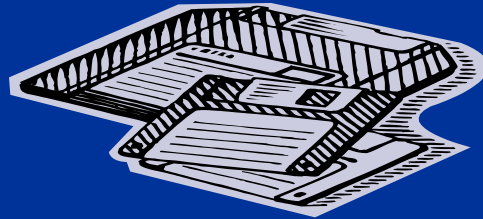
Training Agenda

- The Persistence Problem.
- What is ORM
- What is Hibernate
- Hibernate Architecture
- setup Hibernate Working Environment
- Configuring Hibernate
- Hibernate example Demonstration
- Hibernate Mapping
 - Simple class mapping
 - Collection mapping
 - Inheritance mapping
- Hibernate Query language
- Criteria Query
- Transaction and Concurrency

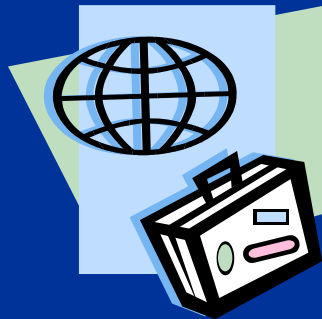
Training Agenda

- Query Optimization techniques
- Hibernate Annotations
- JPA and Hibernate
- Spring and Hibernate
- Conclusion

The Persistence Problem



Object/Relational Mapping



Definition: O/R Mapper

- An O/R mapper bridges between the relational and object models of data.
 - Loads from relational database to objects.
 - Saves from objects to relational database.
- Relational Model: A strict mathematical model of information used by most DBMS packages.
- Object Model: A looser, more familiar information model found in application programming languages.

O/R Mapping

A java class

```
public class Employee {  
    private int id ;  
    private String name;  
    private double sal;  
  
    Employee() { }  
  
    // getter methods  
    .....  
    .....  
    //setter methods  
    .....  
}
```

Employee Table

id	name	sal

The Relational Model - Joins

- Relational entities are associated by joining tables.
 - Fast for arbitrary searching of data.
 - Slow for navigating intrinsic associations.
- This favors a very coarse entity model
 - In other words, keep as much information as possible in one table.

The Object Model

- Object models vary between languages.
 - Not as strict as the relational model.
- In general, though, objects have:
 - Identity
 - State
 - Behavior
- Don't worry about modeling behavior, but identity and state are both critical!

Mapping State

- In general, it's easiest to map an object's state into fields in a table.
- Potential problems:
 - Granularity: object models should generally be finer-grained than relational models.
 - Associations: relationships between entities are expressed very differently
 - Collections or Foreign Keys.

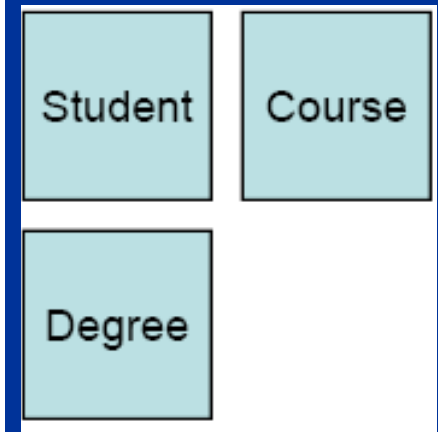
Mapping Identity

- Objects have identity that's independent of their attributes or fields.
- This is foreign to the relational world.
- Solutions:
 - Give up identity and use a natural key.
 - Invent a surrogate key to represent identity.

Approaches to ORM

- Write SQL conversion methods by hand using JDBC
 - Tedious and requires lots of code
 - Extremely error-prone
 - Non-standard SQL ties the application to specific databases
 - Vulnerable to changes in the object model
 - Difficult to represent associations between objects

```
public void addStudent( Student student )
{
String sql = "INSERT INTO student ( name, address )
VALUES ( '" +
Degree
student.getName() + "', '" + student.getAddress() + "' )";
// Initiate a Connection, create a Statement, and execute
the query
}
```

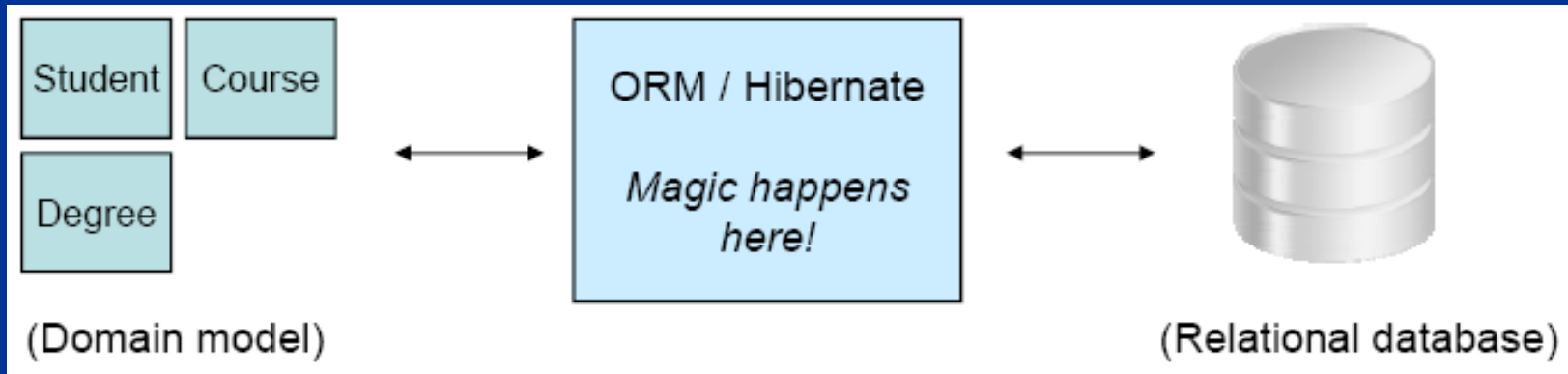


Approaches to ORM

- Use Java serialization – write application state to a file
 - Can only be accessed as a whole
 - Not possible to access single objects
- Object oriented database systems
 - No complete query language implementation exists
 - Lacks necessary features

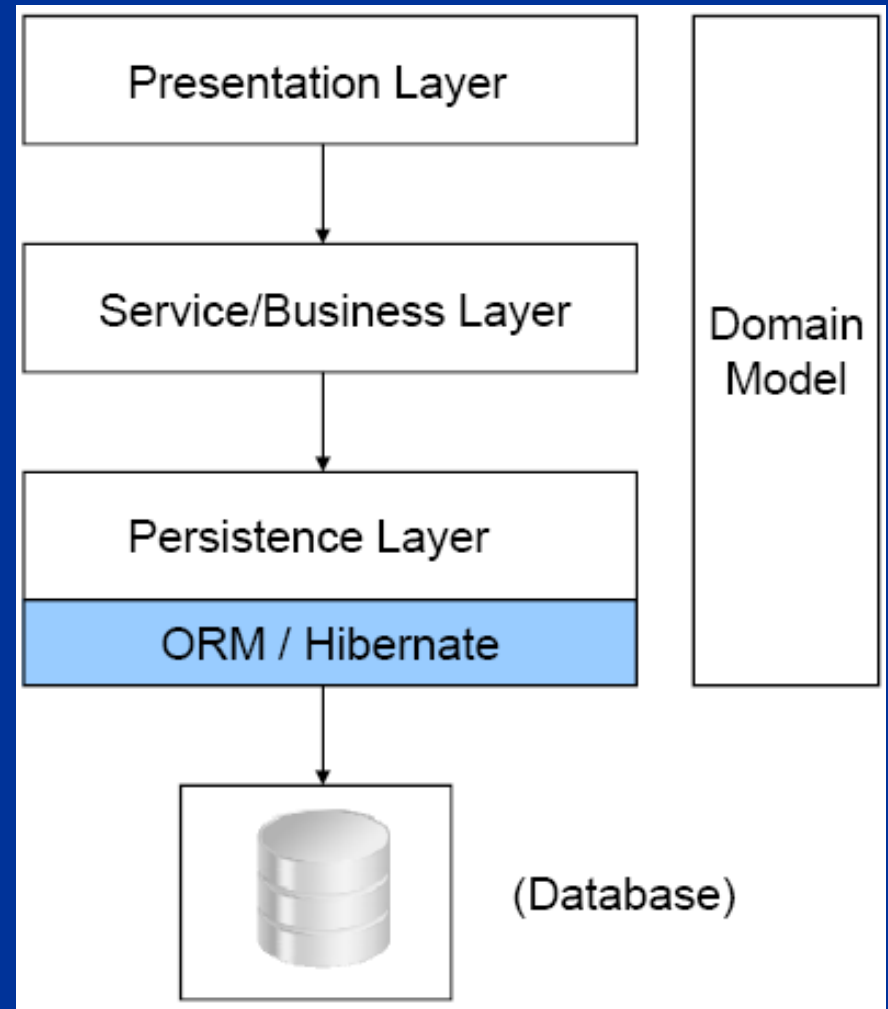
The preferred solution

- Use a *Object-Relational Mapping System* (eg. Hibernate)
- Provides a simple API for storing and retrieving Java objects directly to and from the database
- *Non-intrusive*: No need to follow specific rules or design patterns
- *Transparent*: Your object model is unaware



ORM

- Middleware that manages persistence
- Provides an abstraction layer between the domain model and the database



Hibernate



Hibernate Vital Signs

- Data persistence service for “ordinary” Java objects and classes.
- Associations via Java Collections API.
- O/R mapping defined in an XML file.
- Persistent operations via an external `org.hibernate.Session` object.
- HQL – An object-relational query language.
- Basic tool set for common tasks.

Persistent Classes

- Look a lot like ordinary Java classes.
 - Some special considerations, though.
- Requirements:
 - JavaBeans accessors and mutators.
 - No-argument constructor.
 - Use of Collections interfaces
 - List, not ArrayList.
 - Don't rely on null collections or elements.

Collections for Associations

- Many kinds of collections:
 - Set, Map, List, array, and “Bag” (using List)
 - Even SortedSet and SortedMap.
 - All implemented with foreign keys and/or association tables in the database.
- Hibernate is strict about collections:
 - Sometimes common to use List as a “convenience” for unordered collections.
 - This won't work in Hibernate; Set is better.

Mapping with XML Documents

- Each Hibernate persistent class should have an XML mapping document.
- Defines:
 - Persistent properties.
 - Relationships with other persistent classes.
- Writing these mapping files is the “core” Hibernate persistence task.
 - ... but there are tools for other approaches.

Hibernate Tools

- Plain Hibernate means writing XML mapping.
- Other options include:

hbm2ddl	Generate DDL for database directly from Hibernate mapping files
hbm2java	Generate Java source file directly from Hibernate mapping files
Middlegen	Generate Hibernate mapping from an existing database schema
AndroMDA	Generate Hibernate mapping from UML
XDoclet	Generate Hibernate mapping from annotations in source code

Hibernate Summary

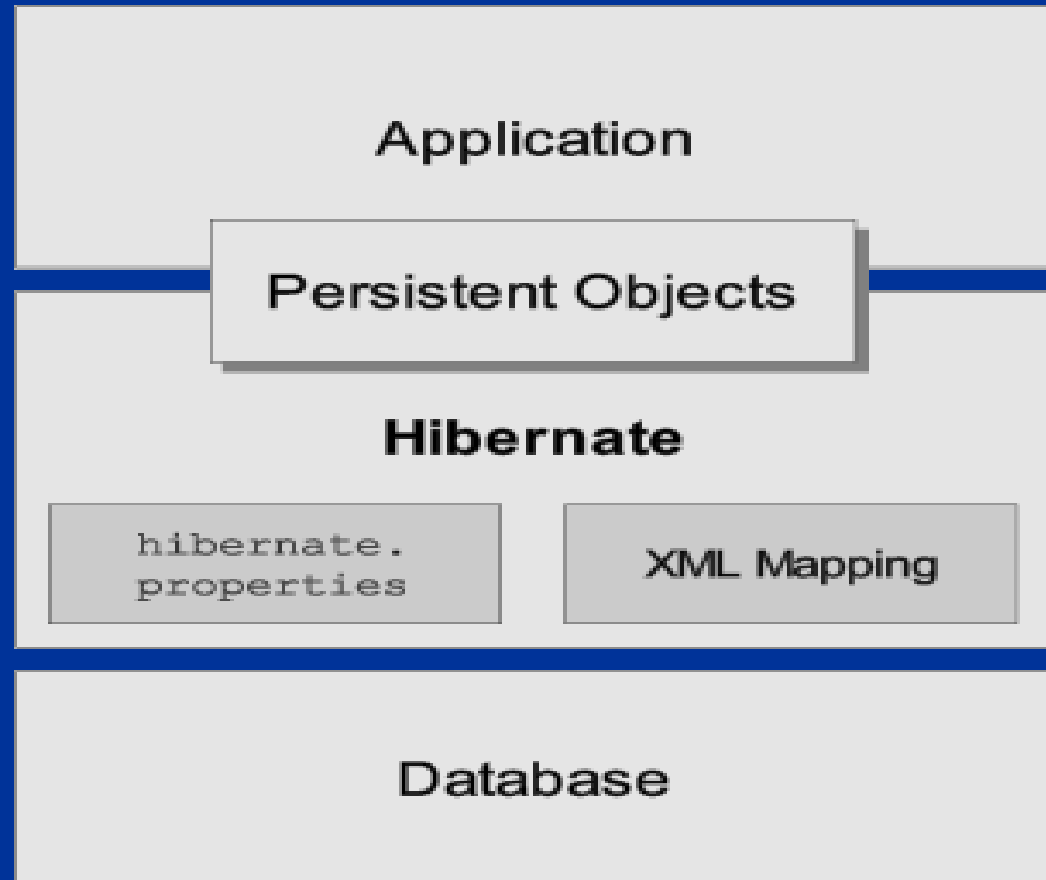
- XML files define the O/R mapping.
- Ordinary objects, but with some rules.
- Associations via the Collections API.
- HQL language for queries.

Hibernate

Architecture

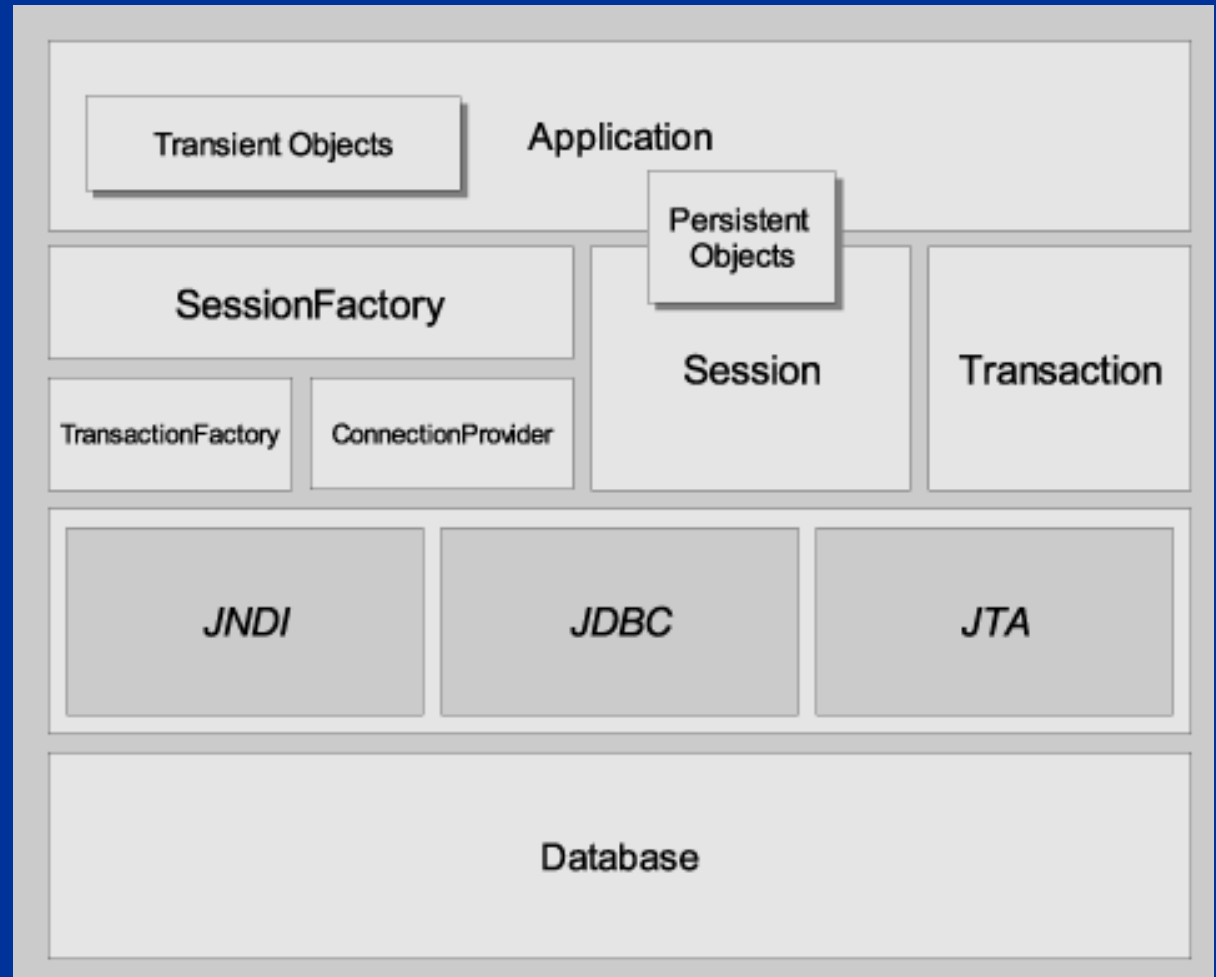
Overview

A (very) high-level
view of the Hibernate
architecture



Detailed Architecture

The Hibernate architecture abstracts the application away from the underlying JDBC/JTA APIs and lets Hibernate take care of the details.



Core Interfaces of Hibernate

- SessionFactory.
- Session.
- Configuration.
- Transaction.
- Query and Criteria.
- Types.

SessionFactory

(org.hibernate.SessionFactory)

- A thread safe (immutable) cache of compiled mappings for a single database.
- A factory for Session and a client of ConnectionProvider.
- Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level.

Session

(org.hibernate.Session)

- A single-threaded, short-lived object representing a conversation between the application and the persistent store.
- Wraps a JDBC connection.
- Factory for Transaction.
- Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.

Configuration

(org.hibernate.cfg.Configuration)

- Used to configure and bootstrap Hibernate.
- A Configuration instance is used to specify the
- location of mapping document and Hibernate specific properties.
- It creates the **SessionFactory**.

Transaction (org.hibernate.Transaction)

- A single-threaded, short-lived object used by the application to specify atomic units of work.
- Abstracts application from underlying JDBC, JTA or CORBA transaction.
- A Session might span several Transactions in some cases. However, transaction demarcation, either using the underlying API or Transaction, is never optional!

Query and Criteria

- Query interface allows to perform queries against the database and control how the query is executed.
- Criteria interface allows to create and execute Object oriented criteria queries.
- A Query instance is lightweight and cannot be used outside the session.

Types

- A Hibernate Type object maps a java type to a database column type.
- All persistent properties of persistent classes, including associations have a corresponding Hibernate type.
- Hibernate has a rich range of built-in-types covering all java primitives and many JDK Classes e.g
 - `java.util.Currency`
 - `java.util.Calendar`
 - `java.io.Serializable`
 - `byte[]`

Working with Hibernate

Hibernate Working Environment

□ Installation

- The hibernate software is available in the form of hibernate.xx.zip file.
- The zip file is unzipped in a suitable location.
- The hibernate package comes with a number of third party jar files which are placed in “<installation-path>hibernate/lib folder.
- All of jar files are not required to compile and run a Hibernate application.

Hibernate Working Environment

- To compile a hibernate application you must place hibernate3.jar in the class path.
- Hibernate uses a number of third party jar file to run which are needed to be in the class path.

Required Hibernate 3rd party .jars

- antlr (required)
 - Hibernate uses ANTLR to produce query parsers, this library is also needed at runtime.
- dom4j (required)
 - Hibernate uses dom4j to parse XML configuration and XML mapping metadata files.
- CGLIB, asm (required)
 - Hibernate uses the code generation library to enhance classes at runtime (in combination with Java reflection).

Required Hibernate 3rd party .jars

- Commons Collections.
- CommonsLogging
 - Required till hibernate version 3.2.x
- EHCache (Optional)
 - Hibernate can use various cache providers for the second-level cache.
 - EHCache is the default cache provider if not changed in the configuration.
- Log4j (optional)
 - Hibernate uses the Commons Logging API, which in turn can use Log4j as the underlying logging mechanism. If the Log4j library is available in the context library directory, Commons Logging will use Log4j

Hibernate Library files

Hibernate 3.2

hibernate3.jar

antlr.jar

cglib-full.jar

asm.jar

asm-attrs.jar

commons-collections.jar

commons-logging.jar

ehcache.jar

jta.jar

dom4j.jar

log4j.jar

Hibernate 3.3

hibernate3.jar

antlr.jar

javaassist

commons-collections.jar

slf4j

ehcache.jar

jta.jar

dom4j.jar

Where to place those .jars?

- Hibernate can use Plain Old java Objects (POJO) for O/R mapping.
- For Stand alone application:
 - Place hibernate3.jar and the third party library files in the classpath.
- For Web applications:
 - Place hibernate3.jar and the third party library files in /WEB-INF/lib folder (in case the application server does not have hibernate support).

Structure of Hibernate Application

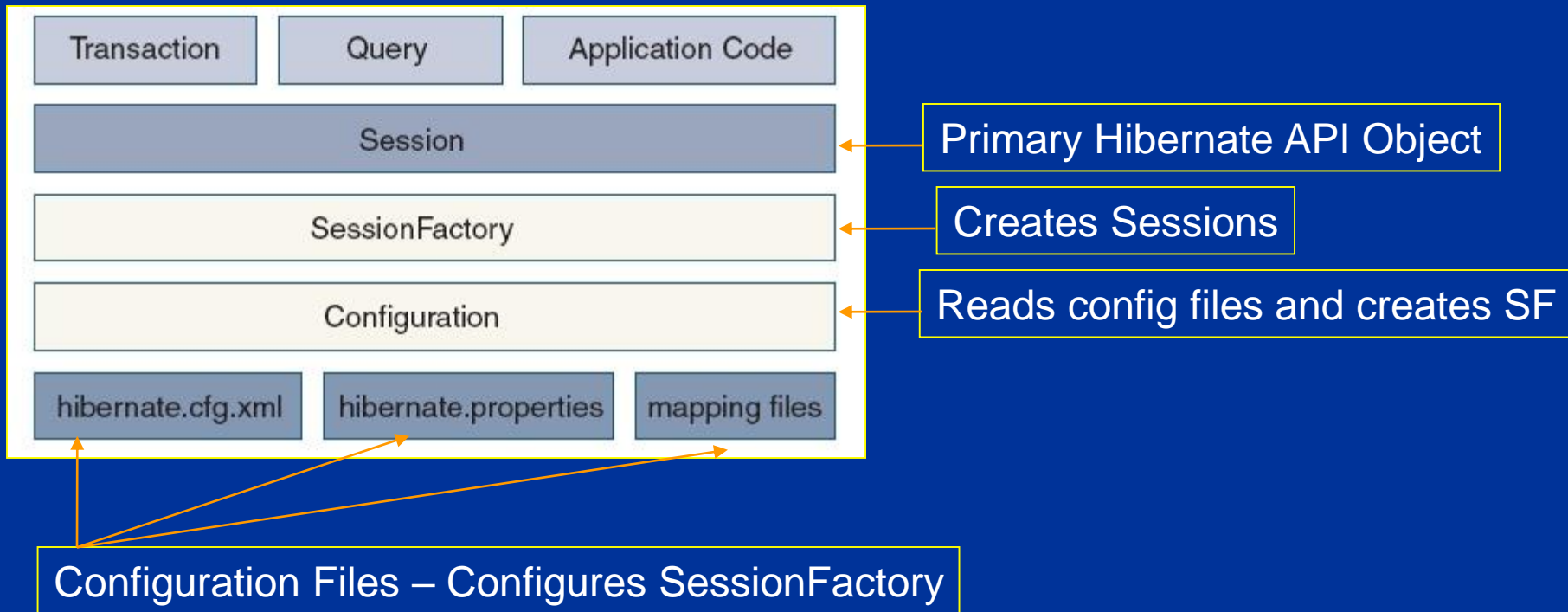
- A Persistent class.
- A configuration file as
 - `hibernate.cfg.xml` OR
 - `hibernate.properties`.(`hibernate.cfg.xml` is preferred)
- A hibernate mapping file.
 - `XX.hbm.xml`

Persistent Classes

- Look a lot like ordinary Java classes.
 - Some special considerations, though.
- Requirements:
 - JavaBeans accessors and mutators.
 - No-argument constructor.
 - Use of Collections interfaces
 - List, not ArrayList.
 - Don't rely on null collections or elements.

Configuring Hibernate

Configuring Hibernate



Configuring Hibernate

- Configuration
 - Setting configuration parameters for Building a SessionFactory.
- An instance of `org.hibernate.cfg.Configuration` represents an entire set of mappings of an application's Java types to an SQL database.
- The Configuration is used to build an (immutable) SessionFactory.
- The mappings are compiled from various XML mapping files.
- Various ways are there to get a Configuration Object.

Configuring Hibernate

- A Configuration instance by instantiating it directly and specifying XML mapping documents. If the mapping files are in the classpath, use `addResource()`

```
Configuration cfg = new  
Configuration()  
  
    .addResource("Item.hbm.xml")  
  
    .addResource("Bid.hbm.xml");
```

- An alternative (sometimes better) way is to specify the mapped class.

```
Configuration cfg = new Configuration()  
  
    .addClass(org.hibernate.auction.Item.class)  
  
    .addClass(org.hibernate.auction.Bid.class);
```

Configuring Hibernate

- A Configuration also allows you to specify configuration properties:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect",
"org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource"
        , "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates",
        "true");
```

Configuring Hibernate

- The Configuration is intended as a startup-time object, to be discarded once a `SessionFactory` is created.
- Configuration parameters are generally written externally using one of the following files:
 - `Hibernate.properties`.
 - `Hibernate.cfg.xml`.
- Preferred approach is to use `hibernate.cfg.xml` for configuration.

hibernate.cfg.xml

- To configure a `hibernate.cfg.xml` file you need a set of mandatory configuration properties known as hibernate jdbc properties
- For Optional tasks you may use some optional properties also, e.g.
 - To show generated sql
 - To auto create database schema
 - To configure a connection pool etc.

Hibernate JDBC Properties

Property name	Purpose
hibernate.connection.driver_class	<i>jdbc driver class</i>
hibernate.connection.url	<i>jdbc URL</i>
hibernate.connection.username	<i>database user</i>
hibernate.connection.password	<i>user password</i>
hibernate.connection.pool_size	<i>maximum number of pooled connections</i>

Configuring hibernate with hibernate.cfg.xml

- Use the following code to configure hibernate using hibernate.cfg.xml

```
Configuration cfg = new Configuration().configure();
```

- The code above looks for a file named hibernate.cfg.xml
- The configure() method is overloaded and a resource from File System, URL and a String as File name can be used

Obtaining a SessionFactory

- Once a Configuration instance is Obtained a SessionFactory is created as:

```
SessionFactory sf =cfg.buildSessionFactory() ;
```

- Hibernate allows an application to instantiate more than one SessionFactory
 - If only one database is used one Sessionfactory is Required.
 - In case more than one database is used one Sessionfactory is required for each Database.
- a <mapping> element contains the address of mapping file

Obtaining a Session and JDBC Connection

- A Session is opened as follows by using the SessionFactory instance

```
Session session = sf.openSession();
```

- As soon as we want to do something using the Session it requires the access to the database and a JDBC connection will be obtained from the pool.
- For this to work, we need to pass some JDBC connection properties to Hibernate through the configuration File.

Example Configuration file

```
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
    oracle.jdbc.driver.OracleDriver
</property>
<property name="connection.url">
    jdbc:oracle:thin:@localhost:1521:xe
</property>
<property name="connection.username">
    hr
</property>
<property name="connection.password">
    hr
</property>
<property name="show_sql">true</property>
<property name="dialect">
    org.hibernate.dialect.Oracle9Dialect
</property>
<!-- Mapping files -->
<mapping resource="User.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

Using the Session – part 2

- Persistent operations are available via a `org.hibernate.Session` object.

<code>Session.delete</code>	Deletes an object from the database
<code>Session.find</code>	Search for objects by an HQL query
<code>Session.filter</code>	Get a subset of some collection
<code>Session.flush</code>	Flush local changes out to the database

Getting started with Hibernate Application

The Entity class

```
package jp.hibernate
class Emp
{
    private String name;
    private int id;
    public int getId()
    {
        return id;
    }
    private void setId(int id)
    {
        this.id = id;
    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name=name;
    }
}
```


The Mapping File (Emp.hbm.xml)

```
<?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>
    <class name="jp.hibernate.Emp" table="EMPLOYEE">
        <id name="id" type="integer" column="ID" >
            <generator class="increment"/>
        </id>
        <property name="name" length="30" type="string" >
            <column name="NAME" />
        </property>
    </class>
</hibernate-mapping>
```

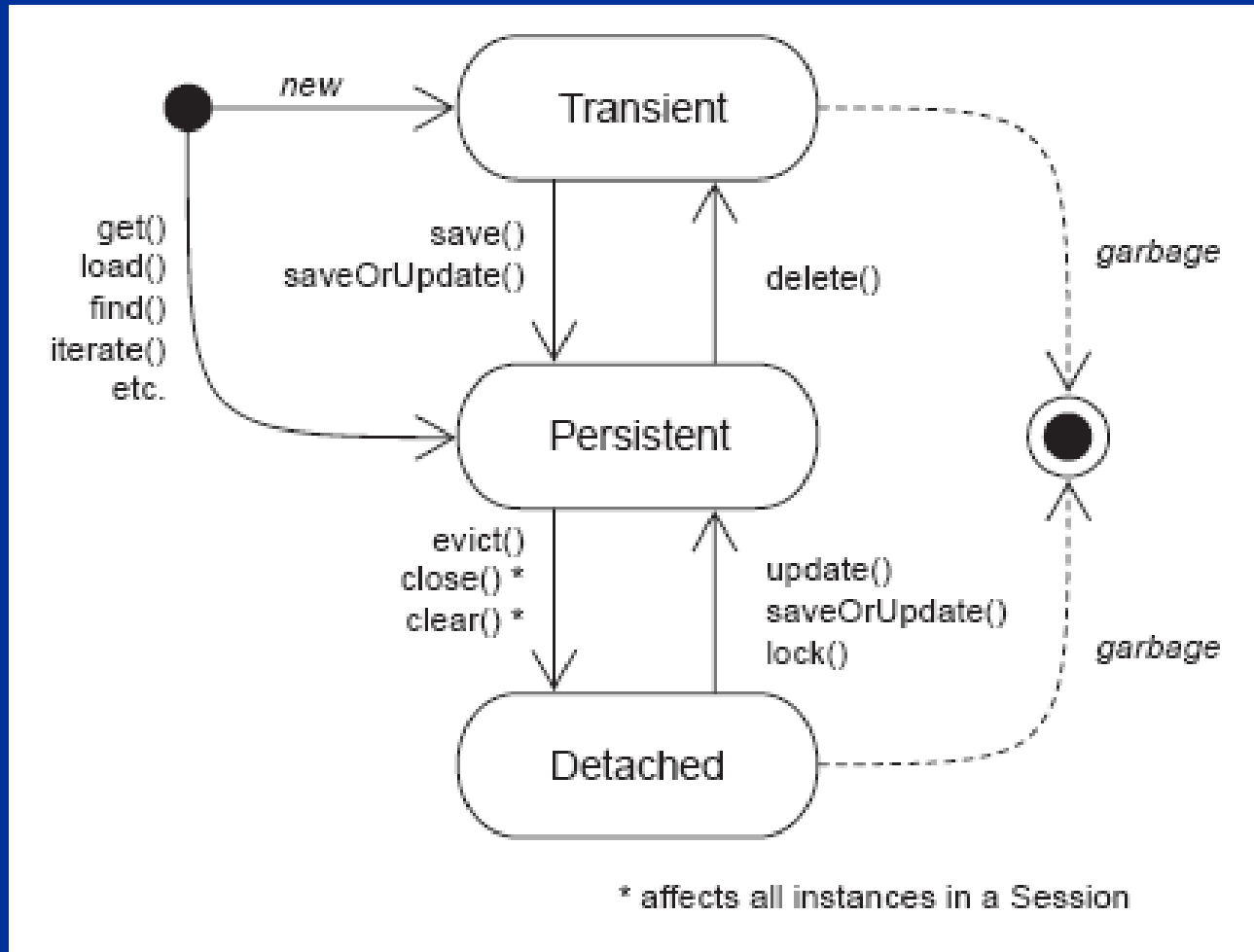
The Configuration File (hibernate.cfg.xml)

```
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
    oracle.jdbc.driver.OracleDriver
</property>
<property name="connection.url">
    jdbc:oracle:thin:@jpserver:1521:jpserver
</property>
<property name="connection.username">
    scott
</property>
<property name="connection.password">
    tiger
</property>
<property name="show_sql">true</property>
<property name="dialect">
    org.hibernate.dialect.OracleDialect
</property>
<!-- Mapping files -->
<mapping resource="emp.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

Test code (code snippet only)

```
class TestName
{
public static void main(String[] args)
{
SessionFactory sf=null;
Session session =null;
Configuration cfg =
    new Configuration().configure();
sf = cfg.buildSessionFactory();
session=sf.openSession();
Transaction tx =
session.beginTransaction();
Emp n = new Emp();
n.setName("Shantanu");
session.save(n);
tx.commit();
session.close();
}
}
```

Entity Lifecycle



Using a Connection pool in hibernate

- ❑ Hibernate uses its default connection pool which is not for production use.
- ❑ In place of the default connection pool C3P0 can be Used.
- ❑ C3P0 is an open source JDBC connection pool.
- ❑ Hibernate will use its C3P0ConnectionProvider for connection pooling if you set hibernate.c3p0.* properties.

A Sample hibernate.cfg.xml file for C3P0 pool

```
<hibernate-configuration>
<session-factory>
.....
<property name="hibernate.c3p0.max_size">
    20
</property>
<property name="hibernate.c3p0.min_size">
    5
</property>
<property name="hibernate.c3p0.timeout">
    1800
</property>
<property name="hibernate.c3p0.acquire_increment">
    20
</property>
.....
</session-factory>
</hibernate-configuration>
```

Configuring Hibernate in a Managed Environment

Sample hibernate.properties for a Container-provided datasource

```
hibernate.connection.datasource =  
                                java:/comp/env/jdbc/datasource  
Hibernate.transaction.factory_class =  
                                org.hibernate.transaction.JTATransactionFactory  
Hibernate.transaction.manager_lookup_class =  
net.sf.hibernate.transaction.JBossTransactionManagerLookup  
Hibernate.dialect = org.hibernate.dialect.OracleDialect
```

JNDI bound SessionFactory

- A SessionFactory should be instantiated only once for an application.
- A single instance should be used by all code in a particular process.
- In a j2EE environment a SessionFactory bound to JNDI is easily shared between different threads.
- The SessionFactory automatically binds itself to the JNDI if the property `hibernate.session_factory_name` is set to the name of the directory node.

JNDI bound SessionFactory

```
hibernate.connection.datasource =  
    java:/comp/env/jdbc/datasource  
hibernate.transaction.factory_class =  
    org.hibernate.transaction.JTATransactionFactory  
hibernate.transaction.manager_lookup_class =  
net.sf.hibernate.transaction.JBossTransactionManagerLookup  
hibernate.dialect = org.hibernate.dialect.OracleDialect  
hibernate.session_factory_name =  
    java:/comp/env/jdbc/datasource  
hibernate.jndi.class =weblogic.jndi.WLInitialContextFactory  
hibernate.jndi.url =t3://localhost:7001
```

O R - Mapping

Hibernate Mapping

- For Persistence Hibernate uses a mapping document, which tells
 - Hibernate how to store the information it finds in the objects and
 - Is an XML File containing the mapping details.
- The format for the mapping document is

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

Example Mapping Document

```
<hibernate-mapping package="com.ibm.hibernate">
<class name="Emp" table="MYEMPLOYEE">
<id name="eid" column="ENO" type="integer">
    <generator class="assigned" />
</id>
<property name="name" column="ENAME" />
<property name="sal" column="SALARY" />
<component name="address" class="Address">
<property name="doorNo" column="R_DOORNO"/>
<property name="city" column="R_CITY"/>
<property name="pin" column="R_PIN">
</component>
</class>
</hibernate-mapping>
```

Important Elements of Mapping Document

- `<hibernate-mapping>`
- `<class>`
- `<property>`
- `<id>`
- `<generator>`
- `<discriminator>`
- `<subclass >`
- `<joined-subclass >`
- `<key >`
- `<column> & <formula>`
- `<import>`
- `<many-to-one>`
- `<one-to-one>`

Note: The elements of mapping will be explained at respective places

<hibernate-mapping>

- This element is the root element for hibernate mapping document
- The element has a number of optional attributes

```
<hibernate-mapping schema="schemaName"  
catalog="catalogName"  
default-cascade="cascade_style" default-  
access="field|property|ClassName" default-lazy="true|false"  
auto-import="true|false" package="package.name" />
```

<class>

- <class> element is the children of <hibernate-mapping>
- Has a number of required attribute as well as optional attributes
- Maps a class to a table

```
<hibernate-mapping package="com.ibm.hibernate">
<class name="Emp" table="MYEMPLOYEE">
<id name="eid" column="ENO" type="integer">
    <generator class="assigned" />
</id>
<property name="name" column="ENAME" />
<property name="sal" column="SALARY" />
-----
</class>
</hibernate-mapping>
```

<id>

- Maps the identifier field of the POJO class to primary key of the mapped table
- Has a <generator> element as child
 - Generator specifies the surrogate key generation strategies offered by hibernate

```
<id name="propertyName" type="typename"
    column="column_name"
    unsavedvalue="null|any|none|undefined|id_value"
    access="field|property|ClassName">
    node="element-name|@attribute-

    name|element/@attribute|"
    <generator class="generatorClass"/>
</id>
```

Note: type refers to a hibernate Type

Understanding Object Identity

- Java object identity and equality
 - Object identity, '==' is notion defined by the java virtual machine.
 - Two Object references are identical (==) if they point to the same memory location.
 - equals() : two objects are equal if they have the same value.
- With object/relational persistence,
 - A persistence object is an in-memory representation of a particular row of a database table.
 - Therefore, along with Java identity and object equality we have database identity.

Understanding Object Identity

- For a persistent Object, we have three methods for identifying Objects:
 - Object identity : Objects occupy the same memory location (==).
 - Object equality : Objects have the same value.
 - Database identity : Objects are identical in a table if they represent the same **row (share the same primary key)**.

<generator>

Class	Description	Types
increment	Creates unique id for column	long,short,int
Identity	DB-supported identity column	long,short,int
sequence	DB-supported sequence	long,short,int
hilo	The hi/lo algorithm generates identifiers that are unique only for a particular database.	long,short,int
seqhilo	Hi/lo using sequence	long,short,int
Uuid.hex	128 bit UUID algorithm incorporating IP address	32 digit hex string
Uuid.string	Same	16 character ASCII string
Native	Varies	best of identity, Sequence or hilo based on database capabilities

<generator>

Class	Description	Types
assigned	Application assigned	Any supported type
guid	uses a database-generated GUID string on MS SQL Server and MySQL	String
Select	retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.	Any supported type
foreign	uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.	Any supported type

<property>

- The <property> element declares a persistent, JavaBean style property of the class

```
<property name="propertyName"
column="column_name"
type="typename"
update="true|false"
insert="true|false"
formula="arbitrary SQL expression" access="field|property|ClassName"
lazy="true|false"
unique="true|false"
not-null="true|false"
optimistic-lock="true|false"
node="element-name|@attribute-name|element/@attribute|."
index="index_name" unique_key="unique_key_id" length="L"
precision="P" precision="S" />
```

Mapping Classes

Objectives

- Mapping a simple java class.
- Component Mapping.
- Mapping collections
 - List.
 - Set.
 - Map.
 - Bag.
 - Idbag.
- Mapping Inheritance.

Mapping a Simple Java Class -- Example

- A class Emp is mapped to table MyEmp

```
<hibernate-mapping
package="com.ibm.hibernate">
<class name="Emp" table="MYEMP">
<id name="eid" column="ENO"
type="integer">
<generator class="hilo" />
</id>
<property name="name">
<column name="ENAME" />
</property>
<property name="sal">
<column name="SALARY" />
</property>
</class>
</hibernate-mapping>
```

```
public class Emp {
    private int eid;
    private String name;
    private double sal;

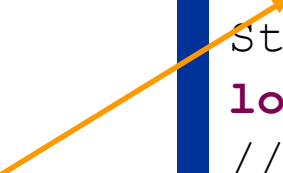
    //setter and getter methods
}
```


Component Mapping

- A component is a contained object that is persisted as a value type, not an entity reference
- The Emp class can be modeled as follows:

```
public class Emp {  
    private int eid;  
    private String name;  
    private double sal;  
    Address address;  
    //setter and getter  
    methods  
}
```

```
public class Address {  
    int doorNo;  
    String city;  
    long pin;  
    //getter and setter methods  
}
```



- Address class does not have an identifier.
 - It is treated as a value type
- Address may be persisted as a component of Emp.

Component Mapping – the mapping file

```
<hibernate-mapping package="com.ibm.hibernate">
<class name="Emp" table="MYEMPLOYEE">
<id name="eid" column="ENO" type="integer">
    <generator class="assigned" />
</id>
<property name="name" column="ENAME" />
<property name="sal" column="SALARY" />
<component name="address" class="Address">
<property name="doorNo" column="R_DOORNO"/>
<property name="city" column="R_CITY"/>
<property name="pin" column="R_PIN"/>
</component>
</class>
</hibernate-mapping>
```

Component Mapping

- Demo and Hands on Lab

Collection Mapping

Collection Mapping

- Hibernate requires that persistent collection-valued fields be declared as an interface type.
 - E.g. `java.util.Set`, `java.util.List`, `java.util.Map` etc.
- Collections may contain almost any other Hibernate type, including all basic types, custom types, components, and of course, references to other entities.
- The contained type is referred to as the *collection element type*.
- Collection elements are mapped by `<element>` or `<composite-element>`, or in the case of entity references, with `<one-to-many>` or `<many-to-many>`.

Mapping a Set

- To map a java.util.Set we use `<set>` element of mapping.
- The collection may be initialized with `HashSet`.

```
public class Item {  
    int itemId;  
    String itemName;  
    String itemDesc;  
    Set<String> images;  
    //setter and getter methods  
}
```

- Item class contains a Set of String Objects.
- Let us see the mapping in the next slide.

Mapping of a Set

```
<hibernate-mapping package="com.ibm.hibernate">
<class name="Item" table="ITEMSTABLE">
<id name="itemId" type="int" column="ITEM_ID">
    <generator class="hilo" />
</id>
<property name="itemName" column="ITEM_NAME" />
<property name="itemDesc" column="ITEM_DESC"/>

<set name="images" table="IMAGES">
<key column="itemId" />
<element type="string" column="FILENAME" />
</set>
</class>
</hibernate-mapping>
```

Mapping of a Set

- The table Structure for the mapping

ITEMSTABLE

ITEM_ID	ITEM_NAME	ITEM_DESC
1	Lamp	It is a designer lamp
2	Nokia phone	It is a mobile phone
3	iPod	Latest iPod

IMAGES

ITEM_ID	FILENAME
1	front.jpg
1	top.jpg
1	back.jpg

Mapping a List

- To map a java.util.List we use `<list>` element of mapping.
- The collection may be initialized with `ArrayList`.

```
public class Item {  
    int itemId;  
    String itemName;  
    String itemDesc;  
    List<String> images;  
    //setter and getter methods  
}
```

- Item class contains a Set of String Objects.
- For every list mapping we need to maintain an index column in the mapped table.

Mapping of a List

```
<hibernate-mapping package="com.ibm.hibernate">
<class name="Item" table="ITEMSTABLE">
<id name="itemId" type="int" column="ITEM_ID">
    <generator class="hilo" />
</id>
<property name="itemName" column="ITEM_NAME" />
<property name="itemDesc" column="ITEM_DESC"/>

<list name="images" table="IMAGES">
<key column="itemId" />
<list-index column="IMAGE_NAME" />
<element type="string" column="FILENAME" />
</list>
</class>
</hibernate-mapping>
```

Mapping of a List

- The table Structure for the mapping

ITEMSTABLE

ITEM_ID	ITEM_NAME	ITEM_DESC
1	Lamp	It is a designer lamp
2	Nokia phone	It is a mobile phone
3	iPod	Latest iPod

IMAGES

ITEM_ID	IMAGE_NAME	FILENAME
1	one	front.jpg
1	two	side.jpg
1	three	top.jpg

Mapping a Map

- Mapping is similar to mapping a List
- To map a java.util.Map we use `<map>` element of mapping.
- The collection may be initialized with `HashMap`.

```
public class Item {  
    int itemId;  
    String itemName;  
    String itemDesc;  
    Map<String> images;  
    //setter and getter methods  
}
```

- Item class contains a Map of String keys and String values.
- For every map mapping we need to maintain an `map-key` column in the mapped table.

Mapping of a Map

```
<hibernate-mapping package="com.ibm.hibernate">
<class name="Item" table="ITEMSTABLE">
<id name="itemId" type="int" column="ITEM_ID">
    <generator class="hilo" />
</id>
<property name="itemName" column="ITEM_NAME" />
<property name="itemDesc" column="ITEM_DESC"/>

<map name="images" table="IMAGES">
<key column="itemId" />
<map-key column="IMAGE_NAME" type="string"/>
<element type="string" column="FILENAME" />
</map>
</class>
</hibernate-mapping>
```

Mapping of a Map

- The table Structure for the mapping

ITEMSTABLE

ITEM_ID	ITEM_NAME	ITEM_DESC
1	Lamp	It is a designer lamp
2	Nokia phone	It is a mobile phone
3	iPod	Latest iPod

IMAGES

ITEM_ID	IMAGE_NAME	FILENAME
1	one	front.jpg
1	two	side.jpg
1	three	top.jpg

Mapping a Bag

- An unordered collection that permits duplicate elements is called a *bag*
- Java does not has Bag or Idbag semantics
- `java.util.Collection` interface can be mapped to a `<bag>` or `<idbag>` element.
- The Collection can be initialised with an `ArrayList`.
- Hibernate supports persistent bags (it uses lists internally but ignores the index of the elements).

Mapping a Idbag

```
private Collection images = new ArrayList();  
...  
public Collection getImages() {  
    return this.images;  
}  
public void setImages(Collection images) {  
    this.images = images;  
}
```

```
<idbag name="images" table="ITEM_IMAGE">  
  <collection-id type="long" column="ITEM_IMAGE_ID">  
    <generator class="sequence"/>  
  </collection-id>  
  <key column="ITEM_ID"/>  
  <element type="string" column="FILENAME" not-  
    null="true"/>  
</idbag>
```


Mapping of Sorted Collection

- A Sorted Map (e.g. `java.util.SortedMap`) is mapped as follows:

```
<map name="images" table="ITEM_IMAGE"
      sort="natural">
  <key column="ITEM_ID" />
  <map-key column="IMAGENAME" type="string" />
  <element type="string" column="FILENAME"
    not-null="true" />
</map>
```

- A `java.util.SortedSet` (with a `java.util.TreeSet` implementation) is mapped as follows:

```
<map name="images" table="ITEM_IMAGE"
      sort="natural">
  <key column="ITEM_ID" />
  <element type="string" column="FILENAME"
    not-null="true" />
</map>
```

Collections of components

Mapping collection of components

- Collections of components are mapped similarly to collections of JDK value type.
- The only difference is the use of <composite-element> instead of an <element> tag.
- The component class may be a POJO ideally implementing hashCode() and equals() method
 - Not necessary for all cases

Mapping a Set of Components

- In previous examples we considered the information of an image to be represented by a String type.
- We may represent the image as a POJO class Image having the following details:

```
public class Image {  
    int imageName;  
    String fileName;  
    int sizeX;  
    int sizeY;  
    //setters and getters  
}
```

Mapping a Set of Components

```
public class Item {  
    int itemId;  
    String itemName;  
    String itemDesc;  
    Set<Image> images;  
    //setters and getters  
}
```

```
<set name="images" table="IMAGES">  
    <key column="ITEM_ID" />  
    <composite-element class="Image">  
        <property name="name" column="IMAGENAME" />  
        <property name="filename" column="FILENAME"/>  
        <property name="sizeX" column="SIZEX" />  
        <property name="sizeY" column="SIZEY" />  
    </composite-element>  
</set>
```

Mapping a Set of Components

- The association from Item to Image is unidirectional. You can navigate to the images by accessing the collection through an Item instance and iterating: `anItem.getImages().iterator()`.
- It may be convenient to access a back pointer like `anImage.getItem()` in some cases
- Hibernate can fill in this property for you if you add a `<parent>` element to the mapping

```
<set name="images" table="IMAGES">
<key column="ITEM_ID" />
<composite-element class="Image">
<parent name="item"/>
<property name="name" column="IMAGENAME" />
<property name="filename" column="FILENAME"/>
<property name="sizeX" column="SIZEX" />
<property name="sizeY" column="SIZEY" />
</composite-element>
</set>
```

<many-to-one>

- An ordinary association to another persistent class is declared using a <many-to-one> element.
- A foreign key in one table is referencing the primary key column(s) of the target table.

```
<many-to-one name="propertyName"  
column="column_name"  
class="ClassName"  
cascade="cascade_style"  
  fetch="join|select"  
update="true|false"  
insert="true|false" />
```

<many-to-one>

```
property-ref="propertyNameFromAssociatedClass"
access="field|property|ClassName"
unique="true|false"
not-null="true|false"
optimistic-lock="true|false"
lazy="true|proxy|false"
not-found="ignore|exception"
entity-name="EntityName"
formula="arbitrary SQL expression"
node="element-name|@attribute-name|element/@attribute|."
embed-xml="true|false" index="index_name"
unique_key="unique_key_id" foreign-key="foreign_key_name"
/>
```


<one-to-one>

```
<one-to-one name="propertyName"  
class="ClassName"  
cascade="cascade_style"  
constrained="true|false"  
fetch="join|select"  
property-ref="propertyNameFromAssociatedClass"  
access="field|property|ClassName"  
formula="any SQL expression"  
lazy="true|proxy|false"  
entity-name="EntityName"  
node="element-name|@attribute-  
name|element/@attribute|." embed-xml="true|false"  
foreign-key="foreign_key_name"  
>
```

Advanced Mapping

- One-to-one.
- Many-to-one.
- Collection mapping.
 - One-to-many.
 - Many-to-many.

Simple Association (one-to-one)

Definition: In this relationship, one first-rank class holds a reference to a single instance of another first-rank class and they are related by a common PK.

Scenario:

The two first-rank classes, Emp and EmpPersonal which are related to each other as follows:

```
EmpPersonal Emp.getEmpPersonal() // returns corresponding
                                // EmpPersonal instance
```

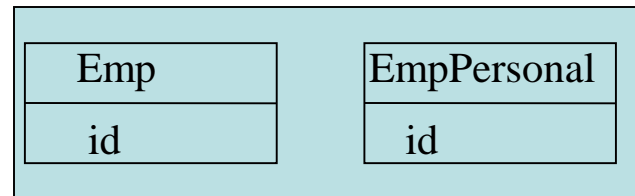
Hibernate Mapping:

```
<class name="Emp" table="Emp"
```

```
...
```

```
<one-to-one name="empPersonal" class="EmpPersonal"/>
```

```
</class>
```



No extra columns are needed to support this relationship. Instead both Emp and EmpPersonal must share the same PK values to be part of a one-to-one association.

If you create suitable instances of Emp and EmpPersonal with a shared PK, then retrieving a Emp will automatically retrieve the corresponding EmpPersonal.

Simple Reference (many-to-one)

Definition: It is the same as the one-to-one situation except there is no requirement to have a shared PK. Instead a FK is used.

Scenario:

We have two first-rank classes, Emp and Dept which are related to each other as follows:

Dept Emp.getDept() // returns corresponding Dept instance

Hibernate Mapping:

In Hibernate, this could be mapped as follows:

```
<class name="Emp" table="Emp">
```

```
...
```

```
<many-to-one name="dept" class="Dept" column="deptno" />
```

```
</class>
```

Emp		Dept
id	deptno	deptno

Now we have created an extra column in Emp table which holds the FK to Dept. Emp and Dept can have completely different PKs and the relationship will still hold.

Bidirectionality: This relationship can be declared both ways, with Bar having *getFoo()*, by simply adding a similar mapping and property to Bar. This will result in Bar's table getting an extra column *foo_id*.

Collection mapping

```
<map name="propertyName"
  table="table_name"
  schema="schema_name"
  lazy="true|false"
  inverse="true|false"
  cascade="all|none|save-update|delete|all-delete-orphan"
  sort="unsorted|natural|comparatorClass"
  order-by="column_name asc|desc"
      where="arbitrary sql where condition"
  fetch="join|select|subselect"
  batch-size="N"
  access="field|property|ClassName"
  optimistic-lock="true|false"
  node="element-name|." embed-xml="true|false"
<key .... />
<map-key .... />
  <element .... />
</map>
```

EMPNO	NAME	SALARY	ADDRESS	DNO
100	Shantanu	30000	Hyd	10
101	Rajiv	30000	Hyd	10
102	Shankar	30000	Blore	20
103	Madhu	30000	Hyd	20
104	Kannan	30000	Dhanbad	20
105	Vimal	30000	Hyd	10
106	Rupa	30000	Bhopal	30
107	Pavan	30000	Hyd	30
108	Kalyan	30000	Chennai	30
109	Bhavesh	30000	Hyd	20
110	Arun	30000	Hyd	20
111	Kanti	30000	Hyd	10
112	Suraj	30000	Hyd	10

*

1

DNO	DNAME	MANAGER
10	Accounts	Rahul
20	IT	Manish
30	HR	Ankur

`<many-to-one name="dept" column="DNO" />`

```
public class Emp {
    int empNo;
    String name;
    String address;
    double salary;
    Dept dept;
}
```

```
public class Dept {
    int deptNo;
    String deptName;
    String manager;
    Set<Emp> emps;
}
```

```
<set name="emps"
table="EMP">
<key column="DNO" />
<one-to-many class="Emp" />
</set>
```

Basic Collection (one-to-many)

Definition: A one-to-many reference is basically a collection. Here a first-rank class, A, holds a reference to a collection of another first-rank class, B.

Scenario:

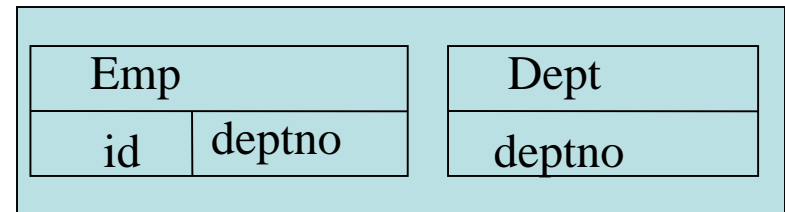
We have two first-rank classes, Foo and Bar which are related to each other as follows:

Set Dept.getEmps() // returns a collection of Emp instances

Hibernate Mapping:

In Hibernate, this could be mapped as follows:

```
<class name="Dept" table="Dept">
...
  <set name="emps" table="Emp">
    <key column="deptno"/>
    <one-to-many class="Emp"/>
  </set>
</class>
```



Bidirectionality: This relationship can be declared both ways, with Emp having geDept(), by suitable code changes to Emp and the following schema change:

```
<class name="Emp" table="Emp">
...
  <many-to-one name="dept" class="Dept" column="deptno"/>
</class>
```

Now your Emps will know who their Dept is. NB: No extra columns are generated for the bidirectionality.

Collection (many-to-many)

Definition: A many-to-many reference is basically a collection. First-rank class A holds a reference to a set of first-rank class B instances (as in the one-to-many case), but B might have multiple A's.

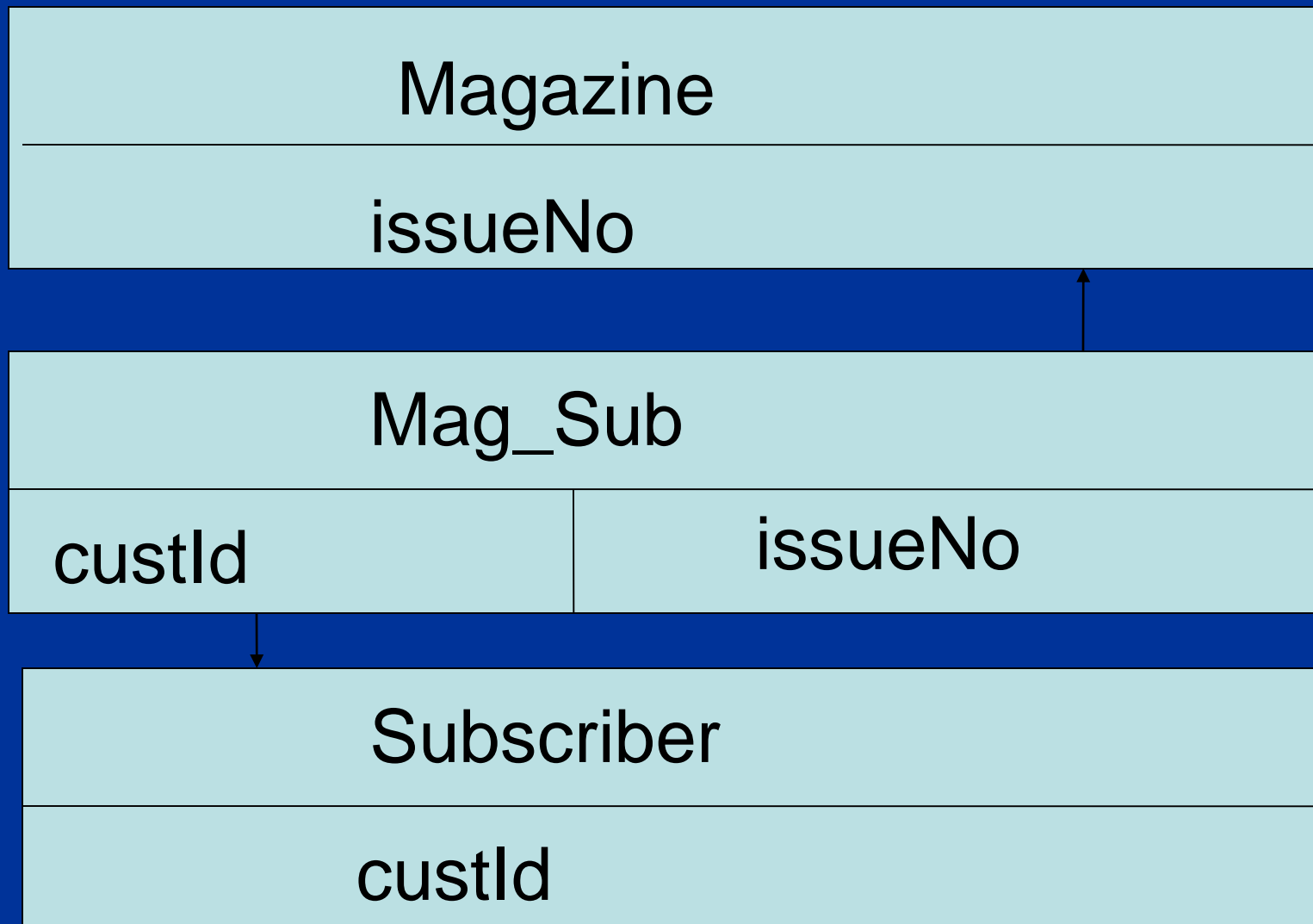
Scenario: We have two first-rank classes, Magazine and Subscriber which are related to each other as follows:

Set Magazine.getSubscribers() // returns a collection of Subscriber instances

Hibernate Mapping: in Hibernate, this could be mapped as follows:

```
<class name="Magazine" table="Magazines">
    ...
    <set name="subscribers" table="Mag_Subs">
        <key column="issueNo"/>
        <many-to-many column="custId" class="Subscriber"/>
    </set>
</class>
```


Collection (many-to-many)



Collection (many-to-many)

In this case we have an extra table, *Mag_sub*, which holds the relationship between instances.

Bidirectionality: This relationship can be declared both ways, with Subscriber having getMagazines(), by suitable code changes to Bar and the following schema change:

```
<class name="Subscriber" table="Subscribers">
  ...
  <set name="magazines" table="Mag_Sub">
    <key column="custId"/>
    <many-to-many column="issueNo" class="Magazine"/>
  </set>
</class>
```

Mapping class Inheritance

- The three different approaches to representing an inheritance hierarchy.
- **Table-per-class hierarchy:** only one table is created containing the columns for all the possible attributes in the base class as well as the child classes. Hibernate tracks data using a discriminator column in the table.
- **Table-per-concrete-class:** one table is created for each concrete class and data is stored in all the tables.
- **Table-per-subclass :** one table for the base class and one table for each subclass is created but subclass table contains the columns for the subclass specific attributes.

<discriminator>

- ❑ Required for polymorphic persistence using the table-per-class-hierarchy mapping
- ❑ Declares a discriminator column of the table.
- ❑ The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row.

```
<discriminator  
column="discriminator_column"  
type="discriminator_type"  
force="true|false"  
insert="true|false"  
formula="arbitrary sql  
expression" />
```

Subclasses

Definition: Subclasses are classes that extend another class in a standard OO inheritance relationship.

Scenario: We have one class, Foo, and another class, Bar, which is a subclass of Foo.

public class Bar extends Foo // Bar is a subclass of Foo

Hibernate Mapping:

In Hibernate, this could be mapped as follows:

```
<class name="Foo" table="foo" discriminator-value="F">
  ...
  <discriminator column="class"/>
  ...
  <subclass name="subclass.Bar" discriminator-value="B">
    <property name="name" column="name" type="string"/>
  </subclass>
</class>
```

Subclasses

Foo		
id	class	name

The *class* field holds a discriminator value. This value tells Hibernate which Java class to instantiate on loading. The subclass, Bar, has its properties stored in the Foo table.

Bidirectionality:

Inheritance relationships are only unidirectional in Java. A child can determine its parent class but the reverse has no meaning.

Working with persistent Objects

- ORM advocates a design in which there are more persistent classes than tables.
- One row represents multiple objects.
- Its persistent state depends on its associated entity.
- As database identity is implemented by primary key, some persistent object won't have their identity.
- One of the objects represented in the row has its own identity and others depend on that.

Entity and value types

□ Entity type

- Has its own database identity (Primary key).
- An object reference to an entity is persisted in as a reference in the DB.
- An entity has its own lifecycle, it may exist independently.

□ Value type

- Has no database identity.
- It belongs to an Entity.

Querying the Database

- Hibernate represents the underlying database using simple java Objects.
- Hibernate facilitates direct retrieval of persistent Objects from the database.
- Hibernate uses the following ways to retrieve Objects from the databases:
 - **Hibernate Query Language (HQL).**
 - **Query by Criteria (QBC).**
 - **Query by Example using Criteria API.**
 - **Native SQL Query.**

HQL – Hibernate Query Language

- HQL is used for building queries to find or filter data from the database.
 - Looks like SQL's SELECT at first glance.
- Differences from SQL:
 - It's only used for searching, not updating.
 - It understands inheritance polymorphism, and object-oriented ownership of associations.
 - Most pieces (even the SELECT clause) are optional in at least some situations!

HQL Syntax

- HQL Syntax and features are very similar to SQL.
- AN HQL Query may consist of
 - Clauses.
 - Aggregate functions.
 - Sub queries.

Before we proceed

We should know that..

- Query and Criteria interfaces both define methods for controlling the execution of a query.
- Query provides methods for binding concrete values to query parameters.
- To execute a Query we need to obtain an instance of one of these interfaces using Session

```
Query q = session.createQuery("from users");
```

Projection

- The class `org.hibernate.criterion.Projections` is a factory for Projection instances. We apply a projection to a query by calling

```
setProjection(). List results = session.createCriteria(Cat.class)  
.setProjection( Projections.rowCount() ).add( Restrictions.eq("color",  
Color.BLACK) ) .list();
```

```
List results = session.createCriteria(Cat.class) .setProjection(  
Projections.projectionList() .add( Projections.rowCount() )  
.add( Projections.avg("weight") ) .add( Projections.max("weight") )  
.add( Projections.groupProperty("color") ) ) .list();
```

- There is no explicit "group by" necessary in a criteria query.
- Certain projection types are defined to be grouping projections, which also appear in the SQL group by clause.

Projection - 2

- An alias may optionally be assigned to a projection, so that the projected value may be referred to in restrictions or orderings.
- Here are two different ways to do this:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"),
        "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

Projection - 3

- The `alias()` and `as()` methods simply wrap a projection instance in another, aliased, instance of `Projection`.
- As a shortcut, you can assign an alias when you add the projection to a projection list:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
    .add( Projections.rowCount(), "catCountByColor" )
    .add( Projections.avg("weight"), "avgWeight" )
    .add( Projections.max("weight"), "maxWeight" )
    .add( Projections.groupProperty("color"), "color" ) )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

Filtering Data

- A Hibernate filter is a global, named, parameterized filter that may be enabled or disabled for a particular Hibernate session.
- A filter criteria is the ability to define a restriction clause very similar to the existing "where" attribute available on the class and various collection elements.
- Except these filter conditions can be parameterized.
- The application can then make the decision at runtime whether given filters should be enabled and what their parameter values should be.
- Filters can be used like database views, but parameterized inside the application.

Filtering Data Syntax

- In order to use filters, they must first be defined and then attached to the appropriate mapping elements.
- To define a filter, use the `<filter-def/>` element within a `<hibernate-mapping/>` element:

```
<filter-def name="myFilter">
```

```
<filter-param name="myFilterParam" type="string"/>
```

```
</filter-def>
```

Interceptors

- The Interceptor interface provides callbacks from the session to the application allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded.
- One possible use for this is to track auditing information.

Interceptors

- For example, the following Interceptor automatically sets the createTimestamp when an Auditable is created and updates the lastUpdateTimestamp property when an Auditable is updated
- A Session-scoped interceptor is specified when a session is opened using one of the overloaded SessionFactory.
- openSession() methods accepting an Interceptor.

*Session session = sf.openSession(new
AuditInterceptor());*

SessionFactory-scoped interceptor

- A SessionFactory-scoped interceptor is registered with the Configuration object prior to building the SessionFactory.
- In this case, the supplied interceptor will be applied to all sessions opened from that Session-Factory; this is true unless a session is opened explicitly specifying the interceptor to use.
- SessionFactory scoped interceptors must be thread safe, taking care to not store session-specific state since multiple sessions will use this interceptor (potentially) concurrently.

new Configuration().setInterceptor(new AuditInterceptor());

Listeners - Event system

- The listeners should be considered effectively singletons; meaning, they are shared between requests, and thus should not save any state as instance variables.
- A custom listener should implement the appropriate interface for the event it wants to process and/or extend one of the convenience base classes (or even the default event listeners used by Hibernate out-of-the-box as these are declared non-final for this purpose).

Listeners - Event system

- Custom listeners can either be registered programmatically through the Configuration object, or specified in the Hibernate configuration XML (declarative configuration through the properties file is not supported).

Listener Sample

□ Here's an example of a custom load event listener:

```
public class MyLoadListener implements
    LoadEventListener {
    // this is the single method defined by the
    LoadEventListener interface
    public void onLoad(LoadEvent event,
        LoadEventListener.LoadType loadType)
        throws HibernateException {
        if ( !MySecurity.isAuthorized(
            event.getEntityClassName(), event.getEntityId()
        ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

Listener Configuration

- You also need a configuration entry telling Hibernate to use the listener in addition to the default listener:

```
<hibernate-configuration>
```

```
<session-factory>
```

```
...
```

```
<event type="load">
```

```
<listener class="com.eg.MyLoadListener"/>
```

```
<listener
```

```
  class="org.hibernate.event.def.DefaultLoadEventListener"/>
```

```
</event>
```

```
</session-factory>
```

```
</hibernate-configuration>
```


Hibernate LazyInitializationException

- LazyInitializationExceptions occur when Hibernate tries to get data from the database, but there is no longer a Session open
- Basically, where you are trying to access the other data Hibernate wants to run a query against the database, but it can't get a Connection to the database, which it gets from the Session object, which is no longer open or available

What is Hibernate Proxy?

- An object proxy is just a way to avoid retrieving an object until you need it.
- Hibernate 2 does not proxy objects by default.
- However, experience has shown that using object proxies is preferred, so this is the default in Hibernate 3

What is Hibernate Proxy?

- Proxies are placeholders that are generated at runtime. Whenever Hibernate returns an instance of an entity class, it checks whether it can return a proxy instead and avoid a database hit
- A proxy is a placeholder that triggers the loading of the real object when it's accessed for the first time:

```
Item item = (Item) session.load(Item.class, new  
                                         Long(123));
```

```
item.getId();
```

```
item.getDescription(); // Initialize the proxy
```

- The third line in this example triggers the execution of the SQL that retrieves an Item into memory
- As long as you access only the database identifier property, no initialization of the proxy is necessary.

Where proxy is helpful?

- A proxy is useful if you need the `Item` only to create a reference, for example:

```
Item item = (Item) session.load(Item.class, new  
    Long(123));
```

```
User user = (User) session.load(User.class, new  
    Long(1234));
```

```
Bid newBid = new Bid("99.99");
```

```
newBid.setItem(item);
```

```
newBid.setBidder(user);
```

```
session.save(newBid);
```

- You first load two objects, an `Item` and a `User`.
Hibernate doesn't hit the database to do this: It returns two proxies. This is all you need, because you only require the `Item` and `User` to create a new `Bid`

How to initialize the Proxy Objects.

- A proxy is initialized if you call any method that is not the identifier getter method, a collection is initialized if you start iterating through its elements or if you call any of the collection-management operations, such as `size()` and `contains()`.
- Hibernate provides an additional setting that is mostly useful for large collections; they can be mapped as *extra lazy*. For example, consider the collection of bids of an Item:

```
<class name="Item" table="ITEM">
```

```
...
```

```
<set name="bids" lazy="extra" inverse="true">
```

```
<key column="ITEM_ID"/>
```

```
<one-to-many class="Bid"/>
```

```
</set>
```

```
</class>
```

Proxy and Lazy fetch

- Setting **lazy="true"** is a shorthand way of defining the persistent class as the proxy. Let's assume the Location class is defined as lazy:
- `<class name="Location" lazy="true"...>...</class>`

Custom SQL for CRUD operations

~~Hibernate gives you the ability to override every single SQL statement generated.~~

We have seen native SQL query usage already, but you can also override the SQL statement used to load or change the state of entities.

```
@Entity
@Table(name="CHAOS")
    @SQLInsert( sql="INSERT INTO CHAOS(size, name, nickname,
id)
VALUES(?,upper(?,?,?,?,?)" @SQLUpdate( sql="UPDATE CHAOS
SET size = ?, name = upper(?),
nickname = ? WHERE id = ?")
    @SQLDelete( sql="DELETE CHAOS WHERE id = ?")
    @SQLDeleteAll( sql="DELETE CHAOS")
    @Loader(namedQuery = "chaos")
    @NamedNativeQuery(name="chaos", query="select id, size, name, lower(
nickname )
as nickname from CHAOS where id= ?", resultClass = Chaos.class)
public class Chaos {
    @Id
    private Long id;    private Long size;    private String name;    private
String nickname;
```

CRUD Configuration

- ❑ @SQLInsert, @SQLUpdate, @SQLDelete, @SQLDeleteAll respectively override the INSERT statement, UPDATE statement, DELETE statement, DELETE statement to remove all entities.
- ❑ Follows the following steps for developing the CRUD application in JPA :
- ❑ The hibernate.cfg.xml is the configuration file for the datasource in JPA. This configuration file contains
 - ❑ (a.) database connection setting (database driver (com.mysql.jdbc.Driver), url (jdbc:mysql://192.168.10.83/hibernateannotation), username (deepak) and password (deepak)),
 - (b.) SQL dialect (dialect - org.hibernate.dialect.MySQLDialect),

CRUD Configuration

(c.) enable hibernate's automatic session context management
(`current_session_context_class - thread`),

(d.) disable the second level cache (`cache.provider_class - org.hibernate.cache.NoCacheProvider`),

(e.) print all executed SQL to stdout (`show_sql - true`) and

(f.) drop and re-create the database schema on startup (`hbm2ddl.auto - none`).

CRUD – SNIPPET part 1

The following is the example for the configuration file.

```
<?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>
```

CRUD – SNIPPET part 2

```
<!-- Database connection settings -->
```

```
<property  
name="connection.driver_class">com.mysql.jdbc.Driver</p  
roperty>
```

```
<property  
name="connection.url">jdbc:mysql://192.168.10.83/hiberna  
teannotation</property>
```

```
<property  
name="connection.username">deepak</property>
```

```
<property  
name="connection.password">deepak</property>
```

```
<!-- JDBC connection pool (use the built-in) -->
```

CRUD – SNIPPET part 3

```
<property name="connection.pool_size">1</property>
<!-- SQL dialect -->
<property
name="dialect">org.hibernate.dialect.MySQLDialect</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.NoCachePro
vider</property>
<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>
```

CRUD – SNIPPET part 4

```
<!-- Drop and re-create the database schema on startup -->
```

```
<property name="hbm2ddl.auto">none</property>
```

```
<mapping class="src.roseindia.Student"/>
```

```
</session-factory>
```

```
</hibernate-configuration>
```

Stored Procedures for querying

- Hibernate 3 introduces support for queries via stored procedures and functions.
- Most of the following documentation is equivalent for both. The stored procedure/function must return a resultset as the first out-parameter to be able to work with Hibernate.

Stored Procedure - Example

- An example of such a stored function in Oracle 9 and higher is as follows:

```
CREATE OR REPLACE FUNCTION selectAllEmployments
RETURN SYS_REFCURSOR
AS
st_cursor SYS_REFCURSOR;
BEGIN
OPEN st_cursor FOR
SELECT EMPLOYEE, EMPLOYER,
STARTDATE, ENDDATE,
REGIONCODE, EID, VALUE, CURRENCY
FROM EMPLOYMENT;
RETURN st_cursor;
END;
```

Stored Procedure - Example

- To use this query in Hibernate you need to map it via a named query.

```
<sql-query name="selectAllEmployees_SP" callable="true">
<return alias="emp" class="Employment">
<return-property name="employee" column="EMPLOYEE"/>
<return-property name="employer" column="EMPLOYER"/>
<return-property name="startDate" column="STARTDATE"/>
<return-property name="endDate" column="ENDDATE"/>
<return-property name="regionCode" column="REGIONCODE"/>
<return-property name="id" column="EID"/>
<return-property name="salary">
<return-column name="VALUE"/>
<return-column name="CURRENCY"/>
</return-property>
</return>
{ ? = call selectAllEmployments() }
</sql-query>
```


Limitations for using stored procedures

- ❑ Stored procedure queries can't be paged with `setFirstResult()/setMaxResults()`.
- ❑ Recommended call form is standard SQL92: { ? = call `functionName(<parameters>)` } or { ? = call `procedureName(<parameters>)` }.
- ❑ Native call syntax is not supported.
- ❑ For Oracle the following rules apply:
 - ❑ A function must return a result set. The first parameter of a procedure must be an OUT that returns a result
 - ❑ Native SQL
 - ❑ set. This is done by using a `SYS_REFCURSOR` type in Oracle 9 or 10. In Oracle you need to define a `REF CURSOR` type.

Limitations for using stored procedures

- For Sybase or MS SQL server the following rules apply:
- The procedure must return a result set. Note that since these servers can/will return multiple result sets and update counts, Hibernate will iterate the results and take the first result that is a result set as its return value.
- Everything else will be discarded.
- If you can enable SET NOCOUNT ON in your procedure it will probably be more efficient, but this is not a requirement

Primary Key

- The <id> element defines the mapping from that property to the primary key column.
- Mapped classes must declare the primary key column of the database table.

```
<id  
name="propertyName"  
type="typename"  
column="column_name"  
unsaved-value="null|any|none|undefined|id_value"  
access="field|property|ClassName">  
node="element-name|@attribute-  
    name|element/@attribute|."  
<generator class="generatorClass"/>  
</id>
```

Primary Key and Object Identity

- Java object identity and equality
 - Object identity, '==' is notion defined by the java virtual machine
 - Two Object references are identical (==) if they point to the same memory location
 - equals() : two objects are equal if they have the same value
- With object/relational persistence,
 - A persistence object is an in-memory representation of a particular row of a database table
 - Therefore, along with Java identity and object equality we have database identity

Primary Key and Object Identity

- For a persistent Object, we have three methods for identifying Objects:
 - Object identity : Objects occupy the same memory location (==)
 - Object equality : Objects have the same value
 - Database identity : Objects are identical in a table if they represent the same **row (share the same primary key)**

Primary Key Generator

- The optional <generator> child element names a Java class used to generate unique identifiers for instances of the persistent class.

```
<id name="id" type="long" column="cat_id">  
  <generator class="org.hibernate.id.TableHiLoGenerator">  
    <param name="table">uid_table</param>  
    <param name="column">next_hi_value_column</param>  
  </generator>  
</id>
```

Composite Key

- Table with a composite key, you may map multiple properties of the class as identifier properties.
- The `<composite-id>` element accepts `<key-property>` property mappings and `<key-many-to-one>` mappings as child elements.

`<composite-id>`

`<key-property name="medicareNumber"/>`

`<key-property name="dependent"/>`

`</composite-id>`

- Your persistent class must override `equals()` and `hashCode()` to implement composite identifier equality.
- It must also implements `Serializable`.

Polymorphic Association

- The <any> mapping element defines a polymorphic association to classes from multiple tables.
- This type of mapping always requires more than one column.
- The first column holds the type of the associated entity.
- The remaining columns hold the identifier.
- It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations.

Polymorphic Association

```
<any name="being" id-type="long" meta-type="string">  
<meta-value value="TBL_ANIMAL" class="Animal"/>  
<meta-value value="TBL_HUMAN" class="Human"/>
```

Querying the Database

- Hibernate represents the underlying database using simple java Objects.
- Hibernate facilitates direct retrieval of persistent Objects from the database.
- Hibernate uses the following ways to retrieve Objects from the databases:
 - **Hibernate Query Language (HQL).**
 - **Query by Criteria (QBC).**
 - **Query by Example using Criteria API.**
 - **Native SQL Query.**

HQL – Hibernate Query Language

- HQL is used for building queries to find or filter data from the database.
 - Looks like SQL's SELECT at first glance.
- Differences from SQL:
 - It understands inheritance polymorphism, and object-oriented ownership of associations.
 - Most pieces (even the SELECT clause) are optional in at least some situations!

HQL Syntax

- HQL Syntax and features are very similar to SQL
- AN HQL Query may consist of
 - Clauses.
 - Aggregate functions.
 - Sub queries.

Before we proceed We should know that..

- Query and Criteria interfaces both define methods for controlling the execution of a query.
- Query provides methods for binding concrete values to query parameters.
- To execute a Query we need to obtain an instance of one of these interfaces using Session

```
Query q = session.createQuery("from users");
```

HQL - Basics

- Simplest possible HQL query:

- `from Employee`

- Returns all employees in the database

- HQL implements the four ANSI join types, plus Oracle-style Cartesian joins.

- Clauses:

- `SELECT` (optional)

- `FROM` (required, except with `Session.filter`)

- `WHERE` (optional)

- Other: `ORDER BY`, `GROUP BY`, `HAVING`, ...

'from' Clause

- Simplest form of an HQL Query.
- Specifies the Object whose instances are to be returned as the query result.
- Commonly used with `SELECT`.
- Syntax:

`From object [as object_alias]`

- `Object_alias` simply means another name given to refer to an object for convenience.
- example

`'from Users as u'`

Will return all the instances of Object **Users**

'select' Clause

- Specifies Object and properties to be returned.
- Used in conjunction with the 'from' clause.
- Syntax:

Select [**object.**]property

e.g: select dept.mgr from comp.Dept as dept

Will return all values of mgr in all instances of dept.

‘where’ Clause

- ‘where’ clause narrows down the search of data.
- ‘where’ clause is used to put a condition during query.

```
List servers = session.createQuery("from Server as server where  
server.port = 0");
```

Listing and Iterating Results

- List() method executes the query and returns the results as a list:

```
List result = session.createQuery('from User').list();
```

- Iterate() method also can be used to iterate through the results:

```
Iterator I = session.createQuery("from User").iterate();
```

Using Named Parameter

- Hibernate supports Named Parameters.
- Using Named parameter the value of a parameter can be set dynamically

```
Query q = session.createQuery(  
    "from Group group wher  
    group.name=:groupname");  
    .setString("groupname","alt.mygroup");  
Iterator I = q.iterate();
```

Important!

- When we pull objects from the database we get a “List” where entries are the actual Objects.
- We can use `get()` to Obtain an Iterator and Type caste the Objects to the required type.
- If HQL is used to pull more than one object then Hibernate returns a List Object containing an array of Objects.

Example

The Query:

```
List stuff = session.createQuery("select  
name,date,sum(records) from table ....");
```

Returns a List of Object arrays. To access each of the values we use code as follows:

```
        Iterator i = stuff.iterator();  
        while(i.hasNext()) {  
            Object[] temp =(Object)i.next();  
String name = (String)temp[0];  
Date date = (Date)temp[1];  
Integer sum =(Integer)temp[2];  
        }
```

Aggregate Functions as supported by HQL

1. `count(*)`: provides a total number of objects persisted in the database.
2. `count(query)`: provides a total no. of objects based on the supplied query
3. `count(distinct attribute)`: provides the count of objects having distinct attribute values
4. `count(attribute)`: provides a count of all the objects in a specified class
5. `avg(attribute)`: provides the average values for the field provided
6. `sum(attribute)`: provides the sum of all values supplied in the attribute
7. `min(attribute)`: provides the minimum of all the supplied attributes
8. `max(attribute)`: provides the maximum of all the supplied attributes

Subselect(subquery)

- ❑ Multiple selection capability is often called “subselect”.
- ❑ The basic idea is to perform a SELECT on the database and then use the result of this SELECT as the input to final SELECT that provides the result.
- ❑ Subselect will be possible only if the underlying database supports it.

```
from Group as group where  
    size(group.stories)>(select  
        avg(storygroup.stories))from Group as storyfroup)
```

HQL Joins

- Hibernate supports the following joins
 - Left Outer Join.
 - Right Outer Join.
 - Inner Join.
 - Full Join.

Left Outer Join

```
Select group.name, count(items)from Group as  
group Left outer join group.stories as items  
group by group.stories.name
```

Obtains the name of a group as well as a count of all stories in the group based on the story name.

Note: left table becomes the source table and all of its rows are included in the result regardless of matching.

Fetch join

- Used to fetch an association eagerly.
- This allows Hibernate to fully initialize an Object along with its association

```
from Group as group left join fetch  
group.stories
```

HQL JOINS

Joins

Person

ID	First_name	Last_name
1	Roger	Whitney
2	Leyland	Beck
3	Carl	Eckberg

Email_address

ID	User_name	host	Person_id
1	Beck	cs.sdsu.edu	2
2	whitney	cs.sdsu.edu	1
3	whitney	rohan.sdsu.edu	1
4	foo	rohan.sdsu.edu	

Joins

- The tables have a column in common as `email_addresses.person_id` refers to `people.id`.
- So we can create a new table by joining the two tables together on that column.

Inner Join (or just Join)

□ Only uses entries linked in two tables

```
select
first_name, last_name, user_name, host
from
people, email_addresses
where
people.id = email_addresses.person_id;
```

First_name	Last_name	User_name	host
Leland	Beck	beck	cs.sdsu.edu
Roger	Whitney	whitney	cs.sdsu.edu
Roger	whitney	whitney	rohan.sdsu.edu

Inner Join (or just Join)

□ Only uses entries linked in two tables

```
select
first_name, last_name, user_name, host
from
people inner join email_addresses
on
(people.id = email_addresses.person_id);
```

First_name	Last_name	User_name	host
Leland	Beck	beck	cs.sdsu.edu
Roger	Whitney	whitney	cs.sdsu.edu
Roger	whitney	whitney	rohan.sdsu.edu

Outer Join and Left Outer Join

- Outer Join
 - Uses all entries from a table
- Left Outer join
 - Use all entries from the left table

Outer Join

```
select  
first_name, last_name, user_name, host  
from  
people left outer join email_addresses  
on  
(people.id = email_addresses.person_id);
```

first_name	last_name	user_name	host
Leland	Beck	beck	cs.sdsu.edu
Roger	Whitney	whitney	cs.sdsu.edu
Roger	Whitney	whitney	rohan.sdsu.edu
Carl	Eckberg		

Right Outer Join

□ Use all entries from the right table
select
first_name, last_name, user_name, host
from
people right outer join email_addresses
on
(people.id = email_addresses.person_id);

first_name	last_name	user_name	host
Leland	Beck	beck	cs.sdsu.edu
Roger	Whitney	whitney	cs.sdsu.edu
Roger	Whitney	whitney	rohan.sdsu.edu
		foo	rohan.sdsu.edu

A right outer join B & B left outer join A

□ The following two statements are equivalent

select

first_name, last_name, user_name, host

from

people **right outer join** email_addresses

on

(people.id = email_addresses.person_id);

select

first_name, last_name, user_name, host

from

email_addresses **left outer join** people

on

(people.id = email_addresses.person_id);

Many-to-One Bidirectional

Classes

```
public class Person {  
    String firstName;  
    String lastName;  
    Set addresses;  
    long id;
```

```
public class EmailAddress {  
    String userName;  
    String host;  
    Person owner;  
    long id;
```

Many-to-One Bidirectional

Tables
People

id	first_name	last_name
1	Roger	Whitney
2	Leland	Beck
3	Carl	Eckberg

Email_Addresses

id	user_name	host	person_id
1	beck	cs.sdsu.edu	2
2	whitney	cs.sdsu.edu	1
3	whitney	rohan.sdsu.edu	1
4	foo	rohan.sdsu.edu	

Many-to-One Bidirectional

```
CREATE TABLE PEOPLE (  
  ID serial NOT NULL ,  
  FIRST_NAME varchar(50) NULL ,  
  LAST_NAME varchar(50) NULL ,  
  CONSTRAINT PEOPLE_PK PRIMARY KEY (id),  
  CONSTRAINT PEOPLE_UNIQ UNIQUE (id))
```

```
CREATE TABLE EMAIL_ADDRESSES (  
  USER_NAME varchar(50) NULL ,  
  HOST varchar(50) NULL ,  
  ID serial NOT NULL ,  
  PERSON_ID int4 NULL ,  
  CONSTRAINT EMAIL_ADDRESSES_PK PRIMARY KEY (id),  
  CONSTRAINT EMAIL_ADDRESSES_UNIQ UNIQUE (id))
```

Many-to-One Bidirectional

```
<hibernate-mapping package="jp.example">
<class name="EmailAddress" table="EMAIL_ADDRESSES" >
<id name="id" type="long" column="id" >
    <generator class="increment"/>
</id>
    <property
        name="userName"
        column="user_name"
        type="string" not-null="false"
        length="50" />
    <property
        name="host"
        column="host"
        type="string" not-null="false"
        length="50" />
</class>
</hibernate-mapping>
```

Many-to-One Bidirectional

```
<hibernate-mapping package="jp.example">
<class name="Person" table="people" >
<id name="id" type="long" column="id" >
<generator class="increment"/>
</id>
<set name="addresses" cascade="all"
table="EMAIL_ADDRESSES">
<key column="person_id"/>
<one-to-many class="jp.example.EmailAddress" />
</set>
<property name="lastName" column="last_name" type="string" not-
null="false"
length="50" />
<property name="firstName" column="first_name" type="string" not-
null="false"
length="50" />
</class>
</hibernate-mapping>
```


Read

```
static void sampleRead() throws MappingException,
HibernateException,
Exception
{
String query = "select p from Person p join p.addresses address
where
address.host like '%gmail%'"
Session session = getHibernateSession();
Query getGmailUser =
session.createQuery(query);
List result = getGmailUser.list();
System.out.println("Number of People: " + result.size());
Object first =result.get(0);
System.out.println( first);
session.close();
}
```

Select

- ❑ `Select address from Person p join p.addresses address where address host like '%gmail%'.`
- ❑ `Select p from Person p join p.addresses address where address host like '%gmail%'.`

Select without a Join

□ `select p from Person p where p.lastName = 'Beck'`

Joins in Hibernate

- Hibernate supports
 - Inner join.
 - Left outer join.
 - Right outer join.

Inner join

- Join defaults to inner
- The following are equivalent

select

p

from

Person p join p.addresses address

□ `//*****//`

select

p

from

Person p inner join p.addresses address

Outer Joins

- The left join will get Eckberg.

*Select p from Person p left join p.addresses address
where address host like '%gmail%' or p.lastName like
'E%'*

- The right join will not get Eckberg.

*Select P from Person p right join p.addresses address
where address host like '%gmail%' or p.lastName like
'E%'*

Selecting Multiple Objects

```
Session session = getHibernateSession();
Query getGmailUser = session
    .createQuery("select p, address from Person p
                  join p.addresses address");
List result = getGmailUser.list();

System.out.println("Number of People: " +
    result.size());
for (int k = 0; k < result.size(); k++)
{
    Object[] resultArray = (Object[]) result.get(k);
    Person personResult = (Person) resultArray[0];
    EmailAddress addressResult
        =(EmailAddress) resultArray[1];
    System.out.println("" + personResult + " " +
        addressResult);
}
session.close();
```

Update

```
Session session = getHibernateSession();

Transaction update = session.beginTransaction();
Query getGmailUser = session.createQuery(
    "select p from Person p join p.addresses address
    where
    address.host='gmail.com'") ;
List result = getGmailUser.list();
Person gmailUser = (Person) result.get(0);
gmailUser.setFirstName("Roger");
Iterator listEmails =
gmailUser.getAddresses().iterator();
while (listEmails.hasNext())
{
    EmailAddress anAddress =(EmailAddress)
    listEmails.next();
    if (anAddress.getHost().equals("gmail.com"))
    anAddress.setHost("gmail.google.com");
}
update.commit();
session.close();
```


Delete

```
Session session = getHibernateSession();
Transaction deleter = null;
try{
    deleter = session.beginTransaction();
    Query getGmailUser = session.createQuery(
        "select p from Person p join p.addresses address
        where
        address.host like '%gmail%'");
    Person result = (Person)
        getGmailUser.uniqueResult();
    session.delete(result);
    deleter.commit();
}
catch (Exception problem)
{
    if (deleter != null) deleter.rollback();
    throw problem;
}
finally
{
    session.close();
}
```

Write, Update & Delete Cascading

```
<set  
name="addresses"  
inverse="false"  
cascade="all"  
table="EMAIL_ADDRESSES">  
<key column="person_id"/>  
<one-to-many class="cs683.EmailAddress" />  
</set>
```

Values for Cascade

- all – cascade write, update & delete.
- none – no cascading.
- save-update – cascade only update & save.
- delete – cascade only deletes.
- delete-orphan.

Query and Native SQL

Query By Criteria

```
session.createCriteria(Category.class)  
.add( Expression.like("name", "Laptop%") );
```

Paging the result

```
Query query =  
session.createQuery("from User u order by u.name asc");  
query.setFirstResult(0);  
query.setMaxResults(10);
```

```
Criteria crit = session.createCriteria(User.class);  
crit.addOrder( Order.asc("name") );  
crit.setFirstResult(40);  
crit.setMaxResults(20);  
List results = crit.list();
```

Paging the result

```
List results =  
session.createQuery("from User u order by u.name asc")  
.setFirstResult(0)  
.setMaxResults(10)  
.list();
```

```
List results =  
session.createCriteria(User.class)  
.addOrder( Order.asc("name") )  
.setFirstResult(40)  
.setMaxResults(20)  
.list();
```

Binding parameters

```
String queryString =  
    "from Item item where item.description like  
        :searchString";  
List result = session.createQuery(queryString)  
    .setString("searchString", searchString)  
    .list();
```

```
String queryString = "from Item item " + "where  
    item.description like :searchString " + "and  
    item.date > :minDate";
```

```
List result = session.createQuery(queryString)  
    .setString("searchString", searchString)  
    .setDate("minDate", minDate)  
    .list();
```


Using positional parameters

```
String queryString = "from Item item "  
+ "where item.description like ? "  
+ "and item.date > ?";
```

```
List result =  
session.createQuery(queryString)  
    .setString(0, searchString)  
    .setDate(1, minDate)  
    .list();
```

Binding arbitrary arguments

- A particularly useful method is `setEntity()`, which lets you bind a persistent entity:

```
session.createQuery("from Item item where  
    item.seller = :seller")  
.setEntity("seller", seller)  
.list();
```

Using named queries

- Hibernate lets you externalize query strings to the mapping metadata, a technique that is called *named queries*.
- This allows you to store all queries related to a particular persistent class (or a set of classes) encapsulated with the other metadata of that class in an XML mapping file.
- The name of the query is used to call it from the application.

Using named queries

- The `getNamedQuery()` method obtains a `Query` instance for a named query:

```
session.getNamedQuery("findItemsByDescription")  
.setString("description", description)  
.list();
```

```
<query name="findItemsByDescription"><![CDATA[  
from Item item where item.description like :description  
]]></query>
```

Named Queries

- Named queries don't have to be HQL strings
- They might even be native SQL queries—and your Java code doesn't need to know the difference

```
<sql-query  
  name="findItemsByDescription"><![CDATA[  
select {i.*} from ITEM {i} where DESCRIPTION  
  like :description  
]]>  
<return alias="i" class="Item"/>  
</sql-query>
```

Named Query

Named SQL Query

```
<sql-query name="persons">
  <return alias="person" class="eg.Person"/>
  SELECT      person.NAME AS {person.name},
              person.AGE AS {person.age},
              person.SEX AS {person.sex}
  FROM PERSON person
  WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
                  .setString("namePattern", namePattern)
                  .setMaxResults(50) .list();
```

Using return-property to explicitly specify column/alias names

- With <return-property> you can explicitly tell Hibernate what column aliases to use

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
    <return-property name="name" column="myName"/>
    <return-property name="age" column="myAge"/>
    <return-property name="sex" column="mySex"/>
</return>
SELECT person.NAME AS myName,
person.AGE AS myAge,
person.SEX AS mySex,
FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```


Using stored procedures for querying

- Hibernate 3 introduces support for queries via stored procedures and functions

```
CREATE OR REPLACE FUNCTION selectAllEmployments
RETURN SYS_REFCURSOR
AS
st_cursor SYS_REFCURSOR;
BEGIN
OPEN st_cursor FOR
SELECT EMPLOYEE, EMPLOYER, STARTDATE, ENDDATE,
REGIONCODE, EID, VALUE, CURRENCY FROM EMPLOYMENT;
RETURN st_cursor;
END;
```

To use this query in Hibernate you need to map it via a named query.

```
<sql-query name="selectAllEmployees_SP" callable="true">
<return alias="emp" class="Employment">
  <return-property name="employee" column="EMPLOYEE"/>
<return-property name="employer" column="EMPLOYER"/>
  <return-property name="startDate" column="STARTDATE"/>
  <return-property name="endDate" column="ENDDATE"/>
  <return-property name="regionCode" column="REGIONCODE"/>
  <return-property name="id" column="EID"/>
  <return-property name="salary">
    <return-column name="VALUE"/>
    <return-column name="CURRENCY"/>
  </return-property>
</return>
{ ? = call selectAllEmployments() }
</sql-query>
```

Come back to Criteria

- Restriction
- For a criteria query, we must construct a Criterion object to express the constraint.
- The Expression class provides factory methods for built-in Criterion types

```
Criterion emailEq = Expression.eq("email",  
    "foo@hibernate.org");  
Criteria crit = session.createCriteria(User.class);  
crit.add(emailEq);  
User user = (User) crit.uniqueResult();
```

Come back to Criteria

- Usually, we would write this a bit less verbosely, using method chaining:

```
User user = (User)
    session.createCriteria(User.class)
        .add( Expression.eq("email", "foo@hibernate.org") )
        .uniqueResult();
```

- The SQL generated by these queries is

```
select U.USER_ID, U.FIRSTNAME, U.LASTNAME,
       U.USERNAME, U.EMAIL
from USER U
where U.EMAIL = 'foo@hibernate.org'
```

Comparison operators Used in HQL

HQL supports the same basic operators as SQL: =, <>, <, >, >=, <=, between, not

between, in, and not in. For example:

from Bid bid where bid.amount between 1 and 10

from Bid bid where bid.amount > 100

from User u where u.email in ("foo@hibernate.org",
"bar@hibernate.org")

Comparison Operators for Criteria Queries

In the case of criteria queries, all the same operators are available via the Expression class:

```
session.createCriteria(Bid.class)
    .add( Expression.between("amount",
new BigDecimal(1),
new BigDecimal(10))
    ).list();
session.createCriteria(Bid.class)
    .add( Expression.gt("amount", new BigDecimal(100) )
    )
    .list();
String[] emails = { "foo@hibernate.org",
    "bar@hibernate.org" };
session.createCriteria(User.class)
    .add( Expression.in("email", emails) )
    .list();
```

String matching

```
from User u where u.firstname like "G%"  
session.createCriteria(User.class)  
.add( Expression.like("firstname", "G%") )  
.list();
```

Or

```
session.createCriteria(User.class)  
.add( Expression.like("firstname", "G",  
MatchMode.START) )  
.list();
```

Logical operators

- Logical operators (and parentheses for grouping) are used to combine expressions:

```
from User user
```

```
where user.firstname like "G%" and  
      user.lastname like "K%"
```

```
from User user
```

```
where ( user.firstname like "G%" and  
      user.lastname like "K%" )
```

```
or user.email in ( "foo@hibernate.org",  
                  "bar@hibernate.org" )
```


Ordering query results

- All query languages provide a mechanism for ordering query results
- HQL provides an ***order by clause***, similar to SQL.
- This query returns all users, ordered by username:

```
from User u order by u.username
```

- You specify ascending and descending order using ***asc*** or ***desc***:

```
from User u order by u.username desc
```

- Finally, you can order by multiple properties:

```
from User u order by u.lastname asc,  
u.firstname asc
```

Ordering query results

□ The Criteria API provides a similar facility:

```
List results =  
    session.createCriteria(User.class)  
        .addOrder( Order.asc("lastname") )  
        .addOrder( Order.asc("firstname") )  
        .list();
```

Batch Operations

Batch Insert

- Set the Hibernate batch size in hibernate.cfg.xml as

```
hibernate.jdbc.batch_size 20
```

- When making new objects persistent, you must flush() and then clear() the session regularly, to control the size of the first-level cache

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<1000000; i++ ) {
    Emp emp = new Emp(.....);
    session.save(emp);
    if ( i % 20 == 0 ) {
        session.flush();
        session.clear();
    }
}
tx.commit();
session.close();
```

Hibernate Type

- Hibernate types map the java types with the database types

<property

name = "email"

column = "EMAIL"

type = "string" />

Here the type= "string" maps a java Stringtype to a SQL varchar/Char type depending upon the dialect chosen

- "string" here is a built in Hibernate type

Hibernate type

- Hibernate type comes with various built in types for O/R mapping.
- Hibernate also provides interfaces UserType.
- By implementing UserType by a Serializable class you can create a custom user type.

Built in Types

Mapping Type	Java Type	Standard Type
integer	Int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
Float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
charcter	Java.lang.String	CHAR(1)
sring	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes_no	boolean or java.lang.Boolean	CHAR(1)(Y or N)
true_false	boolean or java.lang.Boolean	CHAR(1)(T or F)

Built in Types

Date and Time mapping

Mapping type	Java type	Standard SQL Type
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Built in Types

Large Object mapping types

Mapping type	Java type	Standard SQL Type
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

Built in Types

Various JDK Mapping types

Mapping type	Java type	Standard SQL Type
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

Creating UserTypes

- A class which implements Serializable is created
- The class implements hashCode() and equals() methods.
- The class contains the details of user types.
- Now another class is created which implements UserType and handles the above class for custom usertype.

Creating User Types – example code (1/4)

```
public class MonetaryAmount implements Serializable{
    private final BigDecimal value;
    public final    Currency currency;
    public MonetaryAmount(BigDecimal value,Currency
currency) {
        this.value = value;
        this.currency = currency;
    }
    public    BigDecimal getValue() { return value;}
    public    Currency getCurrency() { return currency ;}
    public boolean equals (Object o){ ....; }
    public int hashCode() { .....;}
}
```

Creating User Types – example code (2/4)

```
import org.hibernate.*;
import org.hibernate.auction.model.*;
import org.hibernate.auction.user.UserSession;
import java.math.BigDecimal;
import java.sql.*;
import java.util.Currency;

public class MonetaryAmountSimpleUserType
    implements UserType {

    private static final int[] SQL_TYPES = {Types.NUMERIC};

    public int[] sqlTypes() { return SQL_TYPES; }

    public Class returnedClass() { return MonetaryAmount.class;
    }

    public boolean isMutable() { return false; }

    public Object deepCopy(Object value) {
        return value; // MonetaryAmount is immutable
    }
}
```

Creating User Types – example code (3/4)

```
public boolean equals(Object x, Object y) {
    if (x == y) return true;
    if (x == null || y == null) return false;
    return x.equals(y);
}

public Object nullSafeGet(ResultSet resultSet,
                          String[] names, Object owner)
    throws HibernateException, SQLException {

    if (resultSet.isNull()) return null;
    BigDecimal valueInUSD =
resultSet.getBigDecimal(names[0]);
    Currency userCurrency
        =(Currency) UserSession.get("currency");
    return new MonetaryAmount(valueInUSD,
                              userCurrency);
}
```

Creating User Types – example code (4/4)

```
public void nullSafeSet(PreparedStatement statement, Object
value, int index)
throws HibernateException, SQLException {
    if (value == null) {
        statement.setNull(index, Types.NUMERIC);
    } else {
        MonetaryAmount anyCurrency = (MonetaryAmount) value;
        MonetaryAmount amountInUSD
        = MonetaryAmount.convert( anyCurrency,

            Currency.getInstance("USD") );
        statement.setBigDecimal(index,
            amountInUSD.getValue());
    }
}
```

Transaction

Transaction

- Hibernate supports
 - JDBC Transaction.
 - JTA Transaction.
- Transaction management is exposed to the developer via Hibernate Transaction interface.
- In a standalone or web based application only JDBC Transaction is available.
- In an application Server hibernate can use JTA.

Hibernate transaction API

- The Transaction interface provides methods for declaring demarcation of a database transaction
- `session.beginTransaction()` method marks the beginning of a database transaction
 - in a non managed environment it starts a JDBC transaction on the JDBC connection.
 - in managed environment it starts an new JTA transaction or joins the existing JTA transaction.

Hibernate transaction API

- `commit()` method of the Transaction API synchronizes the session state with the database and commits the underlying Transaction.
- If the Transactional context throws an exception the the Transaction is rolled back using `rollback()` method of the Transaction API.

Example (snippet)

```
Session = sessionFactory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    .....
    tx.commit();
} catch (Exception e) {
    if (tx != null) {
        try {
            tx.rollback();
        } catch (HibernateException he) {
        }
    }
    throw e;
} finally {
    try {
        session.close();
    } catch (HibernateException he) {
        throw he;
    }
}
```

Configuring Transaction

- Hibernate uses the following two *properties* for transaction management
 - `hibernate.transaction.factory_class.`
 - `hibernate.transaction.manager_lookupclass.`

Transaction Factory class

□ To use JTA in managed environment use the factory class

`net.sf.hibernate.transaction.JTATransactionFactory.`

□ To use JDBC Transaction use the factory class

`net.sf.hibernate.transaction.JDBCTransactionFactory`

Hibernate will use the transaction facilities found within the JDBC Driver.

Transaction Manger lookup class

□ The value to the property

`hibernate.transaction.manager_lookupclass`

has to be set based on the application server being used in the application

Application Server	Lookup class
JBoss	<code>net.sf.hibernate.transaction. JBossTtransactionManagerLookup</code>
WebLogic	<code>net.sf.hibernate.transaction. WeblogicTtransactionManagerLookup</code>
WebSphere	<code>net.sf.hibernate.transaction. WebsphereTtransactionManagerLookup</code>

Isolation Levels

- Hibernate provides the following isolation levels
 - Read uncommitted.
 - Read committed.
 - Repeatable read.
 - Serializable.

Read Uncommitted

- Permits dirty reads.
- One transaction may not write to a row if another uncommitted transaction has already written to it.
- This isolation can be implemented using exclusive write locks.

Read Committed

- Permits unrepeatable reads but not dirty reads.
- Achieved using momentary shared locks and exclusive write locks.
- Reading transaction don't block other transactions from accessing a row.
- Uncommitted writing transaction blocks all other transactions from accessing the row.

Repeatable read

- Permits neither unrepeatable reads nor dirty reads.
- Phantom reads may occur.
- Reading transactions block writing transactions.
- Writing transaction blocks all other transactions.

Serializable

- Provides the strictest transaction isolation.
- It emulates serial transaction execution as if transactions had been executed one after the other serially rather than concurrently.

Setting an isolation level

- Every JDBC connection to a database uses the database's default isolation level
usually read committed or repeatable read
- In Hibernate the isolation level can be set using the configuration property

`hibernate.connection.isolation`
to a desired level

- Isolation levels
 - 1 – Read uncommitted isolation
 - 2 – read Committed Isolation
 - 3 – Repeatable read isolation
 - 4 – Serializable isolation

Locking

- Hibernate defines Several lock modes
 - **LockMode.NONE** : Don't go to the database unless the object isn't in either cache.
 - **LockMode.READ** : Bypass both levels of cache, and perform a version check to verify the object in memory is the same version as that currently exists in the database.
 - **LockMode.UPGRADE**: Bypass both levels of cache, and perform a version check and obtain a database-level pessimistic lock upgrade lock, if that is supported.
 - **LockMode.UPGRADE_NOWAIT**: same as UPGRADE , but use a SELECT FOR UPDATE NOWAIT on Oracle. This disables waiting for concurrent lock releases, thus throwing a locking exception can't be obtained.
 - **LockMode.WRITE**: Is obtained automatically when Hibernate has written to a row in the current Transaction. This is an internal mode and can't be specified explicitly.

Using Lock modes

- By default load () and get() use LockMode.NONE
- LockMode.READ is most useful with session.lock() and a detached Object

e.g.

```
Item item .... (detached)
Bid bid = new Bid();
item.addBid(bid);
.....
Transaction tx = session.beginTransaction();
session.lock( item, LockMode.READ);
```

Using Lock modes

- The code performs a version check on `item` instance to verify that the database row wasn't updated by another transaction since it was retrieved, before saving the new `bid`.
- By specifying `LockMode` other than `LockMode.NONE` you force Hibernate to bypass both levels of cache and go all the way to the DB.
- Most of the time caching is more useful than pessimistic locking, so we should not use explicit locking unless it is really needed.

Hibernate Cache

Hibernate Cache

- A cache keeps a representation of the current database state close to the application.
- Memory or the disk on the server machine can be used as a the cache.
- Cache is a local copy of the data.
- Cache sits between the application and the database.

Hibernate Cache

- A cache may be used to avoid a database hit whenever
 - The application performs a lookup by identifier (primary key).
 - Persistence layer resolves an association “lazily”.
- It is also possible to cache the results of queries.

Hibernate Cache

**First Level
Cache**

Session

**Cache Concurrency
Strategy**

Query Cache

Cache Provider

Physical cache region

Second level Cache

First Level Cache

- The first level cache is the **Session** itself.
- The cache associated with the Session is of Transactional scope
- The first level cache is mandatory.
- The first level cache cannot be turned off.

Second Level Cache

- Cache is Pluggable.
- Has process or cluster scope.
- A cache Concurrency strategy defines the transaction isolation details for a particular item of data.
- The cache provider represents the physical, actual cache implementation.
- Use of second level cache is optional.
- Can be configured per class or per association basis.
- Hibernate implements a cache for query results that integrates with second level cache.

Using the First Level Cache

- The session cache ensures that when an application requests the same persistent Object in a particular session it gets back the same (identical) Java instance.
- This cache helps avoid unnecessary database traffic.

Using the First Level Cache

- The First level cache ensures that:
 - The persistence layer is not vulnerable to stack overflow in the case of circular references in graph of objects.
 - Changes made in a particular unit of work are always immediately visible to all other code executed inside the same session.
 - All changes made to an object may be written (flushed) safely to the database.

Using the First Level Cache

- Session cache is always on.
- It need not be enabled explicitly.
- Whenever is an object is passed to `save()`, `update()`, `saveOrUpdate()` and whenever an object is retrieved using `load()`, `list()`, `iterate` or `filter()` that object is added to the Session cache.
- When `flush()` is called the state of the object is synchronized with the database.
- if synchronization is not required but the object must be removed from the memory the `session.evict()` can be called.

Using the Second Level Cache

- ❑ A Hibernate Session is a transaction-level cache of persistent data.
- ❑ It is possible to configure a cluster or JVM-level (SessionFactory-level) cache on a class-by-class and collection-by-collection basis.
- ❑ You may even plug in a clustered cache.
- ❑ Be careful. Caches are never aware of changes made to the persistent store by another application (though they may be configured to regularly expire cached data).
- ❑ You may choose a cache implementation by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`.

Using the Second Level Cache

- The cache policy involves setting of the following:
 - Whether the second level cache is enabled.
 - The Hibernate Concurrency Strategy.
 - The cache expiration policies (time out etc.).
 - The physical format of the cache (memory, indexed files, cluster-replicated).

Benefits and caution

- ❑ Not all classes benefit from second level cache.
- ❑ Second level cache is more often disabled.
- ❑ Cache is useful only for read mostly classes.
- ❑ If you have a data which is more often updated than read, don't enable second level cache.
- ❑ Second level cache can be dangerous in systems which share the database with other writing applications.

Setting up second level cache

- Hibernate second level cache is set up in two steps
 - Decide which concurrency to use.
 - Configure cache expiration and physical cache attributes using the cache provider.

Concurrency Strategies

- Concurrency strategy is a mediator
- Responsible for storing items in the cache.
- Retrieving them from the cache.
- There are four concurrency strategies
 - Transactional.
 - Read-write.
 - Non-strict-read-write.
 - Read-only.

Setting up second level cache

- Hibernate provides built in support for the following cache providers

Cache	Provider class	Type
EHCache	net.sf.hibernate.ehcache.hibernate.Provider	Memory,disk
OSCache	net.sf.hibernate.cache.OSCacheProvider	Memory,disk
SwarmCache	net.sf.hibernate.cache.SwarmCacheProvider	Clustered
TreeCache	net.sf.hibernate.cache.TreeCacheProvider	Clustered

Setting up second level cache

- The cache in the class level can be set using <cache> tag

```
<cache usage = "transactional | read-write |  
                non-strict-read-write |  
                read-only/">
```


Setting up second level cache

□ In collections cache can be used as

```
<class name = "Survey"
  <cache usage = "read-write"/>
  <set name = "Question"
    <cache usage = "read-only"/>
  </set>
</class>
```

Overview over Hibernate Annotations

- Configuration using annotations
 - Hibernate property file (already implemented).
 - hibernate.cfg.xml (not used).
 - O/R mapping file (part of your task).
 - Java annotations (part of your task).
- Putting it all together
 - Hibernate Util (HibernateUtil.getSessionFactory()).
 - Configuration API.
 - AnnotationConfiguration config = **new AnnotationConfiguration();**
 - config.addAnnotatedClass(<class with annotations>);
 - config.addResource(<O/R mapping>);

Overview over Hibernate Annotations

- Annotations
 - @Entity
 - @Id
 - @GeneratedValue
- Consider the strategy
 - @ManyToOne
 - @ManyToMany
- Consider the target entity
 - @OneToMany
- Consider the target entity
 - @Inheritance
- Consider the strategy

Overview over Hibernate Annotations

- Annotation is implemented in EJB 3.0 persistence framework deployment.
- ORM framework discusses about usage of annotation.

Configuration

- Copy all Hibernate3 core and required 3rd party library files (see lib/README.txt in Hibernate).
- Copy hibernate-annotations.jar, lib/hibernate-comons-annotations.jar and lib/ejb3-persistence.jar from the Hibernate Annotations distribution to your class path as well.

Entities

- Plain old Java objects.
- Created by means of new.
- No required interfaces.
- Have persistent identity.
- May have both persistent and non-persistent state.
- Simple types (e.g., primitives, wrappers, enums,serializables).
- Composite dependent object types (e.g., Address).
- Non-persistent state (transient or @Transient).
- Can extend other entity and non-entity classes.
- Serializable; usable as detached objects in other tiers.
- No need for data transfer objects.

Example: Entity

@Entity

```
public class Customer implements Serializable {  
    @Id protected Long id;  
    protected String name;  
    @Embedded protected Address address;  
    protected PreferredStatus status;  
    @Transient protected int orderCount;  
    public Customer() {}  
    public Long getId() {return id;}  
    protected void setId(Long id) {this.id = id;}  
    public String getName() {return name;}  
    public void setName(String name) {this.name = name;}  
    ...  
}
```

Entity Identity

- Every entity has a persistence identity.
- Maps to primary key in database.
- Can correspond to simple type Annotations.
 - `@Id`—single field/property in entity class.
 - `@GeneratedValue`—value can be generated automatically.
 - Can correspond to user-defined class.

Entity Identity

□ Annotations

- `@EmbeddedId`—single field/property in entity class.
- `@IdClass`—corresponds to multiple Id fields in entity class.
- Must be defined on root of entity hierarchy or mapped superclass.

Entity Relationships

- One-to-one, one-to-many, many-to-many, many-to-one relationships among entities.
- Support for Collection, Set, List, Map types.
- May be unidirectional or bidirectional.
- Bidirectional relationships are managed by application, not container.
- Bidirectional relationships have owning side and inverse side.

Example: Relationships

```
@Entity public class Customer {  
    @Id protected Long id;  
    ... @  
    OneToMany protected Set<Order> orders =  
    new HashSet();  
    @ManyToOne protected SalesRep rep;  
    ... public Set<Order> getOrders() {  
        return orders;  
    }  
    public SalesRep getSalesRep() {return rep;}  
    public void setSalesRep(SalesRep rep) {this.rep = rep;}  
}
```

Example: Relationships

```
@Entity public class SalesRep {  
    @Id protected Long id;  
    ... @  
    OneToMany(mappedBy="rep")  
    protected Set<Customer> customers = new HashSet();  
    ... public Set<Customer> getCustomers() {  
        return customers;}  
    public void addCustomer(Customer customer) {  
        getCustomers().add(customer);  
        customer.setSalesRep(this);  
    }  
}
```

Mapping with EJB3/JPA Annotations

- EJB3 entities are plain POJOs. Actually they represent the exact same concept as the Hibernate persistent entities.
- Their mappings are defined through JDK 5.0 annotations (an XML descriptor syntax for overriding is defined in the EJB3 specification).
- Annotations can be split in two categories, the logical mapping annotations (allowing you to describe the object model, the class associations, etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc).

Mapping with EJB3/JPA Annotations

- EJB3 annotations are in the `javax.persistence.*` package.
- Most JDK 5 compliant IDE (like Eclipse, IntelliJ IDEA and Netbeans) can autocomplete annotation interfaces and attributes for you.

Declaring an entity bean

- Every bound persistent POJO class is an entity bean and is declared using the `@Entity` annotation (at the class level):

`@Entity`

```
public class Flight implements Serializable {
```

```
    Long id;
```

`@Id`

```
    public Long getId() { return id; }
```

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

```
}
```

Declaring an entity bean

- @Entity declares the class as an entity bean (i.e. a persistent POJO class), @Id declares the identifier property of this entity bean.
- The other mapping declarations are implicit.
- This configuration by exception concept is central to the new EJB3 specification and a major improvement.
- The class Flight is mapped to the Flight table, using the column id as its primary key column.

Declaring an entity bean

- Depending on whether you annotate fields or methods, the access type used by Hibernate will be field or property.
- The EJB3 spec requires that you declare annotations on the element type that will be accessed, i.e.
- The getter method if you use property access, the field if you use field access. Mixing EJB3 annotations in both fields and methods should be avoided.
- Hibernate will guess the access type from the position of `@Id` or `@EmbeddedId`.

Defining the table

- **@Table** is set at the class level; it allows you to define the table, catalog, and schema names for your entity bean mapping.
- If no **@Table** is defined the default values are used: the unqualified class name of the entity.

@Entity

@Table(name="tbl_sky")

```
public class Sky implements Serializable {
```

```
...
```

Defining the table

- The `@Table` element also contains a schema and a catalog attributes, if they need to be defined. You can also define unique constraints to the table using the `@UniqueConstraint` annotation in conjunction with `@Table` (for a unique constraint bound to a single column, refer to `@Column`).
- `@Table (name="tbl_sky",`
- **`uniqueConstraints =`**
`{@UniqueConstraint(columnNames={"month",`
`"day"})}`
- `)`

Defining the table

- A unique constraint is applied to the tuple month, day. Note that the columnNames array refers to the logical column names.
- The logical column name is defined by the Hibernate NamingStrategy implementation. The default EJB3 naming strategy use the physical column name as the logical column name.
- Note that this may be different than the property name (if the column name is explicit).
- Unless you override the NamingStrategy, you shouldn't worry about that.

Optimistic Locking

- You can add optimistic locking capability to an entity bean using the `@Version` annotation:

`@Entity`

```
public class Flight implements Serializable {
```

```
...
```

`@Version`

`@Column(name="OPTLOCK")`

```
public Integer getVersion() { ... }
```

```
}
```

Optimistic Locking

- The version property will be mapped to the OPTLOCK column, and the entity manager will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).
- The version column may be a numeric (the recommended solution) or a timestamp as per the EJB3 spec.

Optimistic Locking

- Hibernate support any kind of type provided that you define and implement the appropriate `UserVersionType`.
- The application must not alter the version number set up by Hibernate in any way.
- To artificially increase the version number, check in Hibernate Entity Manager's reference documentation `LockMode.WRITE`

Mapping simple properties

□ Declaring basic property mappings

- Every non static non transient property (field or method) of an entity bean is considered persistent, unless you annotate it as `@Transient`.
- Not having an annotation for your property is equivalent to the appropriate `@Basic` annotation.
- The `@Basic` annotation allows you to declare the fetching strategy for a property:
 - `public transient int counter; //transient property`

Mapping simple properties

```
private String firstname; //persistent property
@Transient
String getLengthInMeter() { ... } //transient property
String getName() {... } // persistent property
@Basic
int getLength() { ... } // persistent property
@Basic(fetch = FetchType.LAZY)
String getDetailedComment() { ... } // persistent property
@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... } // persistent property
@Enumerated(EnumType.STRING)
Starred getNote() { ... } //enum persisted as String in database
counter,
```

Declaring column attributes

- The column(s) used for a property mapping can be defined using the `@Column` annotation.
- Use it to override default values (see the EJB3 specification for more information on the defaults). You can use this annotation at the property level for properties that are:
 - not annotated at all
 - annotated with `@Basic`
 - annotated with `@Version`
 - annotated with `@Lob`
 - annotated with `@Temporal`
 - annotated with `@org.hibernate.annotations.CollectionOfElements` (for Hibernate only)

Declaring column attributes

@Entity

```
public class Flight implements Serializable {
```

```
...
```

```
@Column(updatable = false, name = "flight_name",  
        nullable = false, length=50)
```

```
public String getName() { ... }
```

- The name property is mapped to the flight_name column, which is not nullable, has a length of 50 and is not updatable (making the property immutable)

Embedding a component

- To declare an embedded component inside an entity and even override its column mapping.
- Component classes have to be annotated at the class level with the `@Embeddable` annotation.

Embedding a component

- It is possible to override the column mapping of an embedded object for a particular entity using the `@Embedded` and
- `@AttributeOverride` annotation in the associated property:

`@Entity`

```
public class Person implements Serializable {  
    // Persistent component using defaults
```

Embedding a component

```
Address homeAddress;  
@Embedded  
@AttributeOverrides( {  
    @AttributeOverride(name="iso2", column =  
        @Column(name="bornIso2") ),  
    @AttributeOverride(name="name", column =  
        @Column(name="bornCountryName") )  
} )  
Country bornIn;  
...  
}
```

Embedding a component

@Embeddable

```
public class Address implements Serializable {  
    String city;  
    Country nationality; //no overriding here  
}
```

@Embeddable

```
public class Country implements Serializable {  
    private String iso2;  
    @Column(name="countryName") private String name;  
    public String getIso2() { return iso2; }  
    public void setIso2(String iso2) { this.iso2 = iso2; }  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    ... }
```

Non-annotated property defaults

- If a property is not annotated, the following rules apply:
 - If the property is of a single type, it is mapped as `@Basic`
 - Otherwise, if the type of the property is annotated as `@Embeddable`, it is mapped as `@Embedded`
 - Otherwise, if the type of the property is `Serializable`, it is mapped as `@Basic` in a column holding the object in its serialized version
 - Otherwise, if the type of the property is `java.sql.Clob` or `java.sql.Blob`, it is mapped as `@Lob` with the appropriate `LobType`

Mapping identifier properties

- The @Id annotation lets you define which property is the identifier of your entity bean. This property can be set by the application itself or be generated by Hibernate (preferred).
- You can define the identifier generation strategy thanks to the @GeneratedValue annotation:
 - AUTO - either identity column, sequence or table depending on the underlying DB
 - TABLE - table holding the id
 - IDENTITY - identity column
 - SEQUENCE – sequence

Inheritance

- Entities can extend.
- Other entities.
- Either concrete or abstract.
- Mapped super classes.
- Supply common entity state.
- Ordinary (non-entity) Java classes.
- Supply behavior and/or non-persistent state.

Mapping inheritance

- EJB3 supports the three types of inheritance:
- Table per Class Strategy: the <union-class> element in Hibernate.
- Single Table per Class Hierarchy Strategy: the <subclass> element in Hibernate.
- Joined Subclass Strategy: the <joined-subclass> element in Hibernate.
- The chosen strategy is declared at the class level of the top level entity in the hierarchy using the @Inheritance annotation.

Table per class

- `@Entity`
- `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`
- `public class Flight implements Serializable {`
- This strategy support one to many associations provided that they are bidirectional. This strategy does not support the `IDENTITY` generator strategy: the id has to be shared across several tables.
- Consequently, when using this strategy, you should not use `AUTO` nor `IDENTITY`

Single table per class hierarchy

@Entity

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)

@DiscriminatorColumn(
name="planetype",
discriminatorType=DiscriminatorType.STRING
)

@DiscriminatorValue("Plane")

public class Plane { ... }

@Entity

@DiscriminatorValue("A320")

public class A320 extends Plane { ... }

Joined subclasses

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

public class Boat implements Serializable { ... }

@Entity

public class Ferry extends Boat { ... }

Inherit properties from superclasses

```
@MappedSuperclass.  
@MappedSuperclass  
public class BaseEntity {  
    @Basic  
    @Temporal(TemporalType.TIMESTAMP)  
    public Date getLastUpdate() { ... }  
    public String getLastUpdater() { ... }  
    ...  
}  
@Entity class Order extends BaseEntity {  
    @Id public Integer getId() { ... }  
    ...  
}
```

Example: Mapped Superclass

```
@MappedSuperclass public class Person {  
    @Id protected Long id;  
    protected String name;  
    @Embedded protected Address address;  
}  
  
@Entity public class Customer extends Person {  
    @Transient protected int orderCount;  
    @OneToMany  
    protected Set<Order> orders = new HashSet();  
}  
  
@Entity public class Employee extends Person {  
    @ManyToOne  
    protected Department dept;  
}
```


Example: Abstract Entity

```
@Entity public abstract class Person {  
    @Id protected Long id;  
    protected String name;  
    @Embedded protected Address address;  
}  
  
@Entity public class Customer extends Person {  
    @Transient protected int orderCount;  
    @OneToMany  
    protected Set<Order> orders = new HashSet();  
}  
  
@Entity public class Employee extends Person {  
    @ManyToOne  
    protected Department dept;  
}
```

Persistence Context

- Set of managed entity instances at runtime.
- Unique entity identity for any persistent identity.
- Entity instances all belong to same persistence unit; all mapped to same database.
- Persistence unit is unit of packaging and deployment.
- EntityManager API is used to manage. persistence context, control lifecycle of entities, find entities by id, create queries.

Entity Lifecycle

□ New

- New entity instance is created.
- Entity is not yet managed or persistent.

□ Persist

- Entity becomes managed.
- Entity becomes persistent in database on transaction commit.

Entity Lifecycle

□ Remove

- Entity is removed.
- Entity is deleted from database on transaction commit.

□ Refresh

- Entity's state is reloaded from database.

□ Merge

- State of detached entity is merged back into managed entity.

Persist

```
@Stateless public class OrderManagementBean
implements OrderManagement {
...
@PersistenceContext EntityManager em;
...
public Order addNewOrder(Customer customer, Product
product) {
    Order order = new Order(product);
    customer.addOrder(order);
    em.persist(order);
    return order;
}
}
```

Cascading Persist

@Entity

```
public class Customer {
```

```
@Id protected Long id;
```

```
...
```

```
@OneToMany(cascade=PERSIST)
```

```
protected Set<Order> orders = new HashSet();
```

```
}
```

```
...
```

```
public Order addNewOrder(Customer customer, Product  
product) {
```

```
Order order = new Order(product);
```

```
customer.addOrder(order);
```

```
return order;
```

```
}
```

Remove

@Entity

```
public class Order {
```

```
@Id protected Long orderId;
```

```
...
```

```
@OneToMany(cascade={PERSIST,REMOVE})
```

```
protected Set<LineItem> lineItems = new HashSet();
```

```
}
```

```
...
```

```
@PersistenceContext EntityManager em;
```

```
...
```

```
public void deleteOrder(Long orderId) {
```

```
Order order = em.find(Order.class, orderId);
```

```
em.remove(order);
```

```
}
```

Merge

@Entity

```
public class Order {
```

```
    @Id protected Long orderId;
```

```
    ...
```

```
    @OneToMany(cascade={PERSIST, REMOVE, MERGE})
```

```
    protected Set<LineItem> lineItems = new HashSet();
```

```
}
```

```
...
```

```
@PersistenceContext EntityManager em;
```

```
...
```

```
public Order updateOrder(Order changedOrder) {
```

```
    return em.merge(changedOrder);
```

```
}
```


Collections

- You can map Collection, List (ie ordered lists, not indexed lists), Map and Set.
- The EJB3 specification describes how to map an ordered list (ie a list ordered at load time) using `javax.persistence.OrderBy` annotation:
- this annotation takes into parameter a list of comma separated (target entity) properties to order the collection
- by (eg `firstname asc, age desc`), if the string is empty, the collection will be ordered by id.

Java Persistence Query Language

- An extension of EJB™ QL
- Like EJB QL, a SQL-like language
- Added functionality
- Projection list (SELECT clause)
- Explicit JOINS
- Subqueries
- GROUP BY, HAVING
- EXISTS, ALL, SOME/ANY
- UPDATE, DELETE operations
- Additional functions

Projection

```
□ SELECT e.name, d.name  
FROM Employee e JOIN e.department d  
WHERE e.status = 'FULLTIME'  
SELECT new com.example.EmployeeInfo(e.id, e.name,  
e.salary, e.status, d.name)  
FROM Employee e JOIN e.department d  
WHERE e.address.state = 'CA'
```

Subqueries

```
□ SELECT DISTINCT emp  
FROM Employee emp  
WHERE EXISTS (  
SELECT mgr  
FROM Manager mgr  
WHERE emp.manager = mgr  
AND emp.salary > mgr.salary)
```

Joins

```
□ SELECT DISTINCT o
FROM Order o JOIN o.lineItems l JOIN l.product p
WHERE p.productType = 'shoes'

SELECT DISTINCT c
FROM Customer c LEFT JOIN FETCH c.orders
WHERE c.address.city = 'San Francisco'
```

Update, Delete

```
□ UPDATE Employee e
SET e.salary = e.salary * 1.1
WHERE e.department.name = 'Engineering'
DELETE
FROM Customer c
WHERE c.status = 'inactive'
AND c.orders IS EMPTY
AND c.balance = 0
```

Queries

- Static queries.
- Defined with Java language metadata or XML.
- Annotations: `@NamedQuery`, `@NamedNativeQuery`.
- Dynamic queries.
- Query string is specified at runtime.
- Use Java Persistence query language or SQL.
- Named or positional parameters.
- `EntityManager` is factory for Query objects.
- `createNamedQuery`, `createQuery`,
`createNativeQuery`.
- Query methods for controlling max results,
pagination, flush mode.

Dynamic Query

```
□ @PersistenceContext EntityManager em;

...

public List findByZipcode(String personType, int zip) {
return em.createQuery (
“SELECT p FROM ” + personType + “p WHERE
    p.address.zip
= :zipcode”)
.setParameter(“zipcode”, zip)
.setMaxResults(20)
.getResultList();
}
```


Static Query

```
□ @NamedQuery(name="customerFindByZipcode", query  
  = "SELECT c FROM Customer c WHERE  
    c.address.zipcode = :zip")
```

```
@Entity public class Customer {...}
```

```
...  
public List findCustomerByZipcode(int zipcode) {  
  return em.createNamedQuery ("customerFindByZipcode")  
    .setParameter("zip", zipcode)  
    .setMaxResults(20)  
    .getResultList();  
}
```

Object/Relational Mapping

- Map persistent object state to relational database.
- Map relationships to other entities.
- Mapping metadata may be annotations or XML (or both)
- Annotations.
 - Logical—object model (e.g., @OneToMany, @Id, @Transient).
 - Physical—DB tables and columns (e.g., @Table, @Column).
- XML
 - Elements for mapping entities and their fields or properties.
 - Can specify metadata for different scopes.
 - Rules for defaulting of database table and column names.

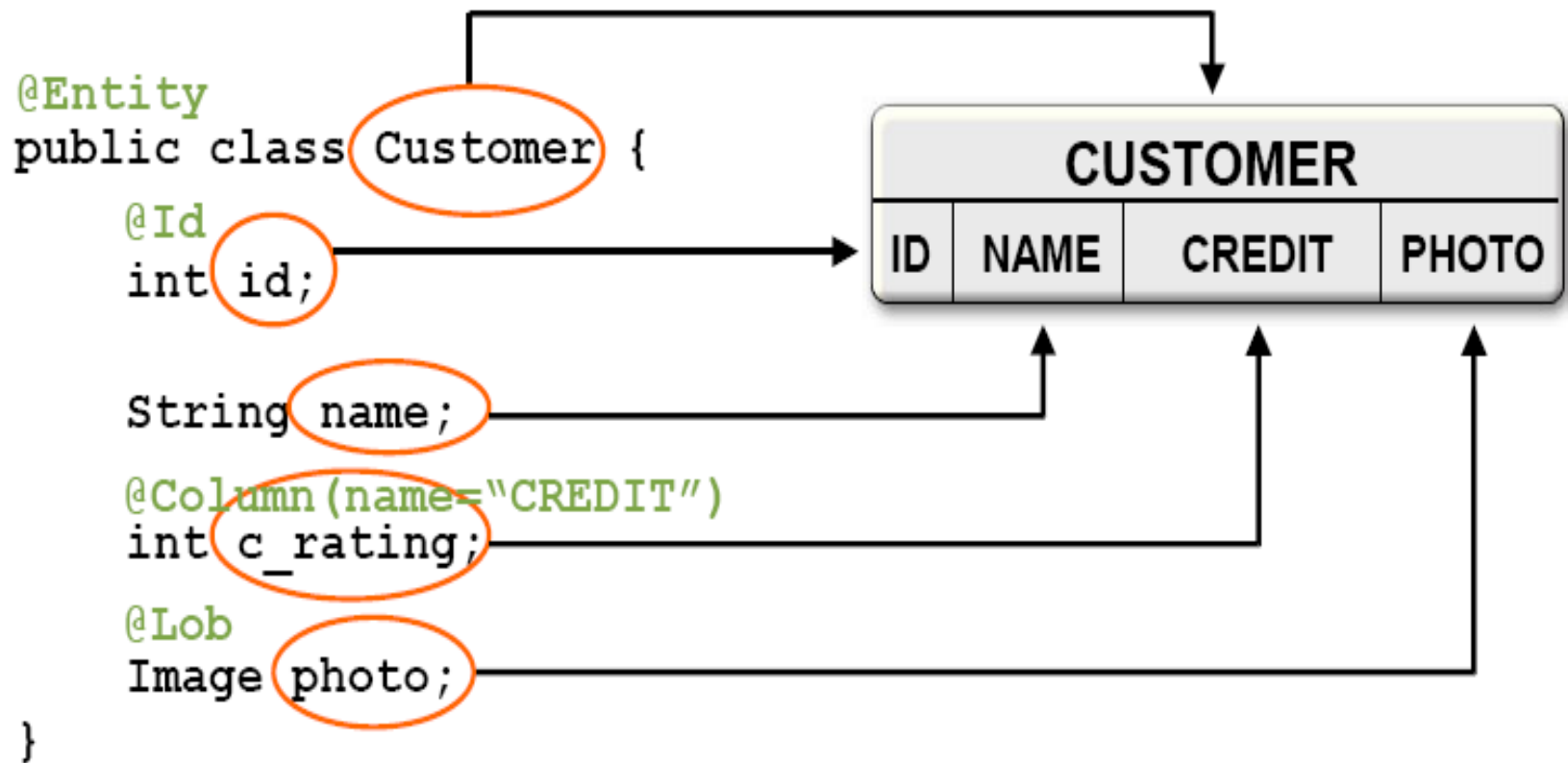
Object/Relational Mapping

- ❑ State or relationships may be loaded or “fetched” as EAGER or LAZY .
- ❑ LAZY is a hint to the Container to defer loading until the field or property is accessed.
- ❑ EAGER requires that the field or relationship be loaded when the referencing entity is loaded.
- ❑ Cascading of entity operations to related entities
- ❑ Setting may be defined per relationship.
- ❑ Configurable globally in mapping file for persistence-by-reachability.

Simple Mappings

- Direct mappings of fields/properties to columns.
- `@Basic`—optional annotation to indicate simple mapped attribute.
- Maps any of the common simple Java types.
- Primitives, wrapper types, `Date`, `Serializable`, `byte[]`, ...
- Used in conjunction with `@Column`.
- Defaults to the type deemed most appropriate if no mapping annotation is present.
- Can override any of the defaults.

Simple Mapping



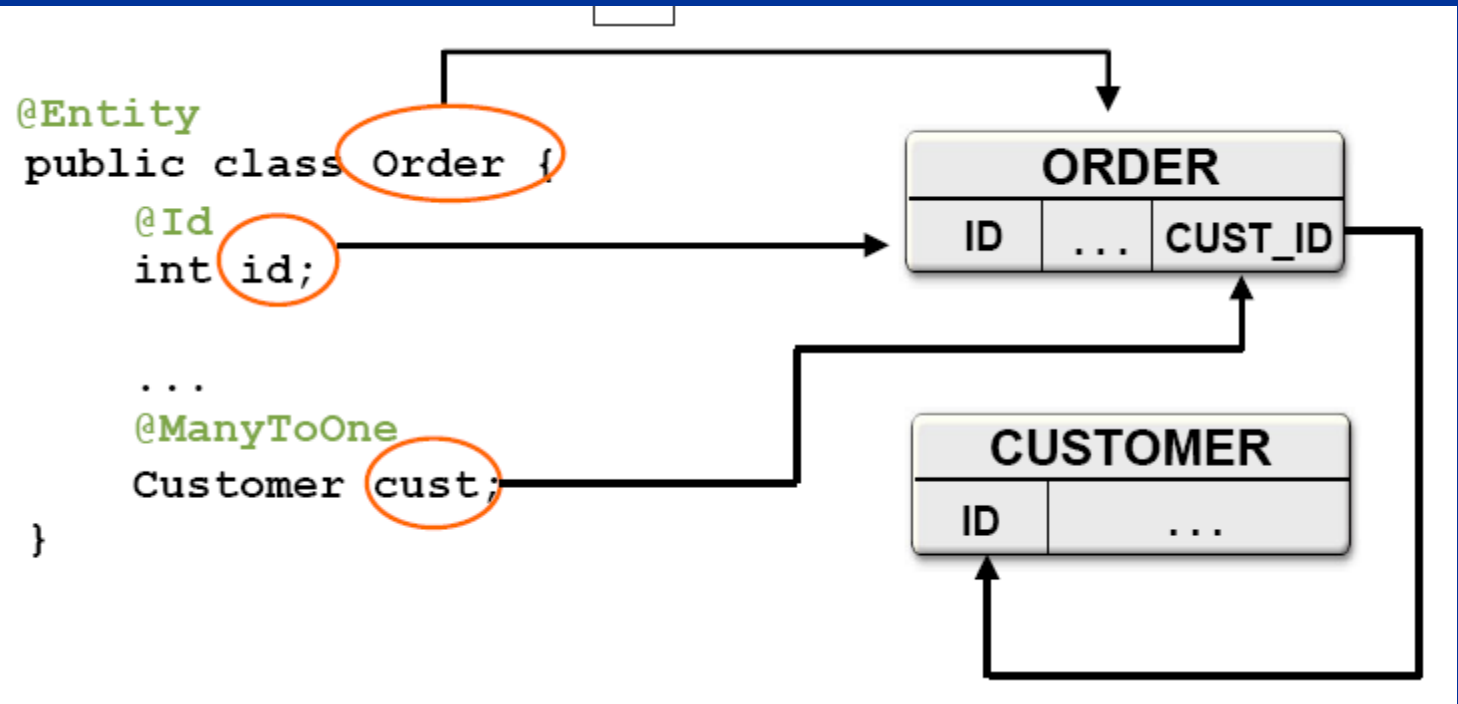
Simple Mappings

```
<entity class="com.acme.Customer">  
  <attributes>  
    <id name="id"/>  
    <basic name="c_rating">  
      <column name="CREDIT"/>  
    </basic>  
    <basic name="photo"><lob/></basic>  
  </attributes>  
</entity>
```

Relationship Mappings

- Common relationship mappings supported.
- @ManyToOne, @OneToOne—single entity.
- @OneToMany, @ManyToMany—collection of entities.
- Unidirectional or bidirectional.
- Owning and inverse sides of every bidirectional relationship.
- Owning side specifies the physical mapping.
- @JoinColumn to specify foreign key column.
- @JoinTable decouples physical relationship mappings from entity tables.

Many-to-One Mapping



Many-to-One Mapping

```
<entity class="com.acme.Order">
```

```
<attributes>
```

```
<id name="id"/>
```

```
...
```

```
<many-to-one name="cust"/>
```

```
</attributes>
```

```
</entity>
```

One-to-Many Mapping

```
@Entity
public class Customer {
    @Id
    int id;
    ...

    @OneToMany(mappedBy="cust")
}

@Entity
public class Order {
    @Id
    int id;
    ...
    @ManyToOne
    Customer cust;
}
```



One-to-Many Mapping

```
<entity class="com.acme.Customer">
```

```
<attributes>
```

```
<id name="id"/>
```

```
...
```

```
<one-to-many name="orders" mapped-by="cust"/>
```

```
</attributes>
```

```
</entity>
```

Many-to-Many Mapping

```
@Entity
public class Customer {
    @Id
    int id;
    ...
    @ManyToMany
    Collection<Phone> phones;
}
```

```
@Entity
public class Phone {
    @Id
    int id;
    ...
    @ManyToMany(mappedBy="phones")
    Collection<Customer> custs;
}
```



Many-to-Many Mapping

```
<entity class="com.acme.Customer">
```

```
<attributes>
```

```
...
```

```
<many-to-many name="phones"
```

```
<join-table name="CUST_PHONE">
```

```
<join-column name="CUST_ID"/>
```

```
<inverse-join-column name="PHON_ID"/>
```

```
</join-table>
```

```
</many-to-many>
```

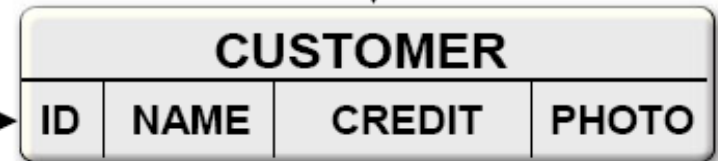
```
</attributes>
```

```
</entity>
```

Embedded Objects

```
@Entity
public class Customer {
    @Id
    int id;
    @Embedded
    CustomerInfo info;
}
```

```
@Embeddable
public class CustomerInfo {
    String name;
    int credit;
    @Lob
    Image photo;
}
```



Embedded Objects

```
<entity class="com.acme.Customer">
```

```
<attributes>
```

```
...
```

```
<embedded name="info"/>
```

```
</attributes>
```

```
</entity>
```

```
<embeddable class="com.acme.CustomerInfo">
```

```
<attributes>
```

```
<basic name="photo"><lob/></basic>
```

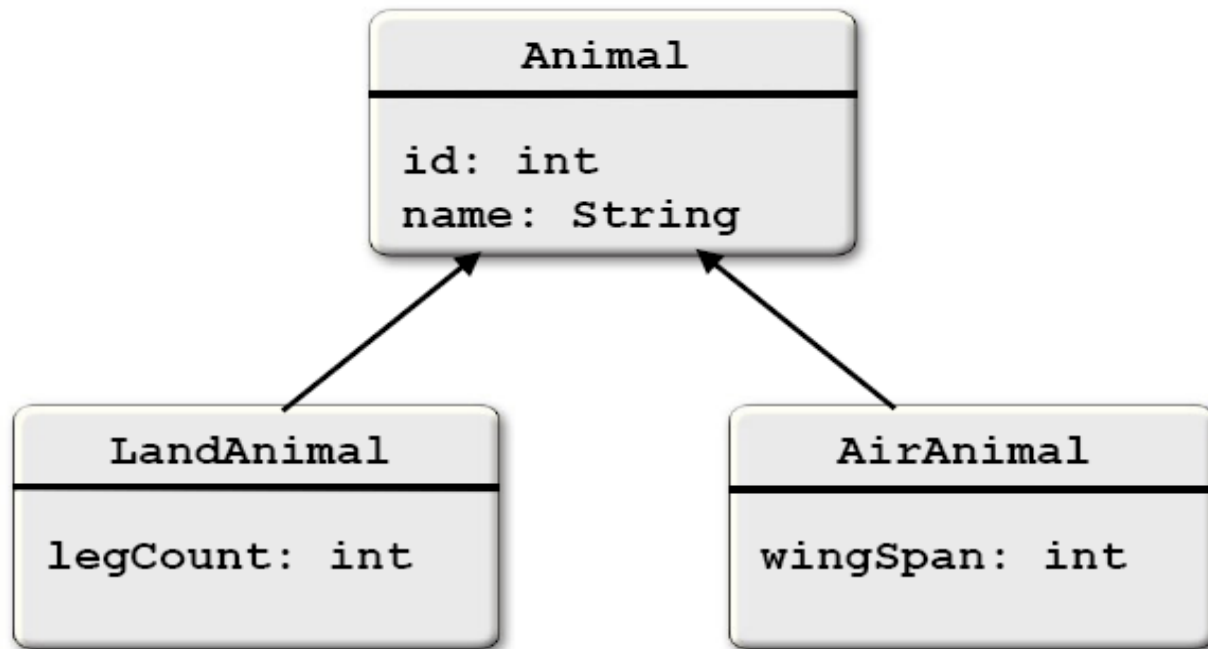
```
</attributes>
```

```
</embeddable>
```

Inheritance

- Entities can extend.
- Other entities—concrete or abstract.
- Non-entity classes—concrete or abstract.
- Map inheritance hierarchies in three ways
 1. SINGLE_TABLE.
 2. JOINED.
 3. TABLE_PER_CLASS.

Object Model



Data Models

- Single table:

ANIMAL				
ID	DISC	NAME	LEG_COUNT	WING_SPAN

- Joined:

ANIMAL	
ID	NAME

LAND_ANIMAL	
ID	LEG_COUNT

AIR_ANIMAL	
ID	WING_SPAN

- Table per Class:

LAND_ANIMAL		
ID	NAME	LEG_COUNT

AIR_ANIMAL		
ID	NAME	WING_SPAN

Persistence in Java SE

- ❑ No deployment phase.
- ❑ Application must use a “Bootstrap API” to obtain an EntityManagerFactory.
- ❑ Typically use resource-local EntityManagers.
- ❑ Application uses a local EntityTransaction obtained from the EntityManager.
- ❑ New persistence context for each and every EntityManager that is created.
- ❑ No propagation of persistence contexts.

Entity Transactions

- Resource-level transaction akin to a JDBC transaction.
- Isolated from transactions in other EntityManagers.
- Transaction demarcation under explicit application control using EntityTransaction API.
- `begin()`, `commit()`, `setRollbackOnly()`, `rollback()`, `isActive()`.
- Underlying (JDBC™) resources allocated by EntityManager as required.

Bootstrap Classes

- Root class for bootstrapping an EntityManager.
- Locates a provider service for a named persistence unit.
- Invokes on the provider to obtain an EntityManagerFactory
 - **javax.persistence.EntityManagerFactory**
- Creates EntityManagers for a named persistence unit or configuration

Example

```
public class SalaryChanger {  
    public static void main(String[] args) {  
        EntityManagerFactory emf = Persistence  
            .createEntityManagerFactory("HRSystem");  
        EntityManager em = emf.createEntityManager();  
        em.getTransaction().begin();  
        Employee emp = em.find(  
            Employee.class, new Integer(args[0]));  
        emp.setSalary(new Integer(args[1]));  
        em.getTransaction().commit();  
        em.close();  
        emf.close();  
    }  
}
```

ORM - JPA - Hibernate

- Most ORM tools in the Java world are based on JDBC.
- Connection configuration easy and similar to JDBC.
- JPA - first common standard released in May 2006 as part of the Java Standard and Java.
- Enterprise Edition 1.5 Based largely on Hibernate.
- Hibernate still exceeds JPA functionality in some areas and remains most popular JPA implementation.

Other JPA implementations

- OpenJPA – an Apache JPA implementation.
- Apache Cayenne – another Apache JPA project.
- JPOX – Open Source JPA implementation.
- TopLink – Oracle's JPA implementation.
- Kodo – BEA's JPA flavour.
- Eclipse Link – recently started project for INGRES.
- Integration – contributions welcome!

JPA Configuration

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">
<persistence-unit name="jpa-ingres">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<properties>
<property name="hibernate.archive.autodetection" value="class"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.connection.driver_class"
value="com.ingres.jdbc.IngresDriver"/>
<property name="hibernate.connection.url" value="jdbc:ingres://localhost:117/
performancedb"/>
<property name="hibernate.dialect"
value="org.hibernate.dialect.IngresDialect"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
</properties>
</persistence-unit>
```

JPA Configuration - Notes

- More than one persistence unit can be defined to cater for multiple databases etc.
- Changes only in configuration file.
- Actual code not touched by implementation.
- SQL Logging is easily configured.
- Hibernate implementation needs a number of Jar files on the class path.

JPA entity configuration

```
package uk.co.luminary.entity;
import javax.persistence.*;
@Entity
@Table(name="EMP_TABLE")
public class Employee
{
    //define the primary key
    @Id
    @GeneratedValue
    (strategy=GenerationType.AUTO)
    private Integer id;
    private String firstName; private String surName; private Date
    birthdate;
    private String position; private String notes; //default constructor
    public Employee(){ } }
```

Id generation strategies in JPA

javax.persistence.GenerationType.AUTO

javax.persistence.GenerationType.IDENTITY

javax.persistence.GenerationType.SEQUENCE

javax.persistence.GenerationType.TABLE

@Id

@SequenceGenerator(name="s1", sequenceName="SEQ")

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="s1")

public long getId() {return id;}

@Id

**@TableGenerator(name="tg", table="pk_table", pkColumnName="name",
valueColumnName="value", allocationSize=10)**

@GeneratedValue(strategy=GenerationType.TABLE, generator="tg")

public long getId() {return id;}

When to use JPA

- Any system using Java and a RDBMS.
- New systems with a complex business logic.
- Existing systems based on JDBC.
- Especially strong: read-heavy systems.
- Although little DB knowledge necessary on Java side, use Ingress expert to make sure DB is well tuned.

Legacy Systems

- When extending – JDBC and JPA can live side by side.
- Ensure clear boundaries to keep code maintainable.
- Decide whether both is necessary – slow step-by-step migration possible?
- Make sure they don't clash.
- Known problems: ID generation – ideally let either JDBC OR JPA create IDs for any one table/entity.

Migrate old JDBC code to JPA?

- Reasoning
 - performance issues.
 - maintainability?
- Many advantages:
 - More maintainable.
 - More readable.
 - Less code to maintain.
 - Potentially more efficient (due to caching etc.).
- Should be done gradually to avoid surprises.
- Extensive testing necessary.
- Can be expensive – replication of JDBC in JPA needs careful re-design.

Optimization Techniques

- Using List () return the results, Hibernate will initialize all of the query results for the persistent object, the result set is larger, will take up a lot of processing time.

Optimization Techniques

- ❑ Iterator.next () return the object and use the object, Hibernate will only call the corresponding query object initialization, for large amount of data, each inquiry will be a call spend more time.
- ❑ When the result set is larger, but contains the same data more substantial, or not all result sets are used, the use of iterator () have advantages.

Optimization Techniques

- And the use of `iterator ()` return results, at each call to
- For large amount of data, use `qry.scroll ()` can get a better deal with the speed and performance.
- But also directly on the result set back rolling forward.

Optimization Techniques

- In the case of associated operations, Hibernate can express complex data relationships, but please be used with caution, so that data will be more easy get a better efficiency, especially at deeper levels of relevance, performance will be poor.
- containing associated PO (persistent object), if the default-cascade = "all" or "save-update", new PO, please pay attention to a collection of PO in the assignment, because there is likely to make an update operation of the implementation of many.

Optimization Techniques

- In one-to-many, many-to-one relationship, use the lazy loading mechanism, make a lot of object will be used at initialization, this can make to save memory space and to reduce the database load, but if the PO The collection has not been used, can reduce the inter-database interaction so as to reduce the processing time.

Optimization Techniques

- In the case of large amount of data to add, modify, or delete operation is a large amount of data queries, and database interactions decide the number of treatment time are the most important factor in reducing the frequency of interaction are the best way to enhance efficiency, so at development process, please `show_sql` set to true, a deeper understanding of the treatment process of Hibernate, try a different way, can make efficiency.

Optimization Techniques

- Hibernate is a JDBC-based, but Hibernate is optimized JDBC, Hibernate one of the use of a buffer mechanism will enhance the performance, such as the use of secondary cache and query cache, if a higher percentage of hits that performance would be to substantially upgrade.
- Hibernate can be set `hibernate.jdbc.fetch_size`, `hibernate.jdbc.batch_size` such as property, optimized for Hibernate.

Optimization Techniques

- It is worth noting, however, a number of databases provided by the primary key generation mechanism may not be the best in terms of efficiency on a large number of concurrent insert data may lead to the interlocking between the table.
- Database provided by the primary key generation mechanism, often at an internal table to preserve the current primary key status (such as for self-by-type primary key, this internal table on maintaining the current maximum and increment), followed by each time data will be inserted to read the maximum, then add incremental volume of the Record as a new primary key, then after the new update to Maximum internal table, so that an Insert operation cause possible multiple tables within the database to read and write operation, at the same time have data associated with locking unlocking operation, which have had a greater impact on performance.
- Therefore, concurrent Insert demanding systems, recommend the use of uuid.hex as the primary key generation mechanism.

Optimization Techniques

- Dynamic Update, if selected, the Update SQL generated does not contain does not happen when changes in the fields property, which would to some extent on upgrading SQL performance.
- Dynamic Insert If selected, the Insert SQL generated does not contain the changes did not happen property fields, so that at a certain extent, enhance SQL performance
- In the preparation of the code when requested, on the POJO's getter / setter method set to public, if set to private, Hibernate will not be able to optimize property access, only to turn the traditional reflex mechanisms operate This will cause many performance overhead (especially in pre-1.4 version of Sun JDK and the IBM JDK, the reflection caused by the system overhead is considerable).

Optimization Techniques

- In one-to-many relationship, a party set to take the initiative will be many side (inverse = false) will help to improve performance.
- Because of poor performance of many-to-many association (because of the introduction of the middle table, a read operation requires repeated several inquiries), and therefore the design of large-scale use should be avoided.

Optimization Techniques

- Hibernate supports two locking mechanisms: that is called "pessimistic locking (Pessimistic Locking)" and "optimistic-lock (Optimistic Locking)".
- Pessimistic lock database performance brought about by the substantial overhead, especially for the long services, such expenses are often not unbearable.
- Optimistic locking mechanism at a certain extent, resolved the problem. Optimistic locking mechanism to avoid long transaction database locking overhead, greatly enhance the volume of a large complicated system under the overall performance.

Integration with Legacy database

- Hibernate, being an Object->Relational DB Mapper for Java, needs access to JDBC connections.
- Out of the box, Hibernate is fairly self contained when it comes to connection control.
- By default, Hibernate ships with the ability to obtain a data source implementation (`javax.sql.DataSource`) from JNDI by setting the properties appropriately:

DataSource

- `hibernate.connection.datasource = java:/comp/env/jdbc/test`
- Alternatively, if JNDI isn't an option, you can use a Hibernate-internal connection pool implementation (C3PO), and simply give driver/url information for Hibernate to create and pool its own connections.

Hibernate-cfg.xml

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:
postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
```

JNDI and Hibernate

- `Session session = sessionFactory.openSession();`
- The data source implementation gives you control over the connections, but it requires JNDI availability.
- The connection pooling implementation doesn't require JNDI, but now Hibernate has its own pool of connections, and you cannot gain access to them (well, you can, but it's a breach of the Hibernate API rules).

Syntax

- Additionally, if you have a need to give Hibernate connections to use on a per-session basis, you can always ask the session factory for a session running on a particular connection:

```
Connection conn = // ... get connection from some outside source.  
Session session = sessionFactory.openSession(conn);
```

Connection Provider

- If you have legacy connection pooling or non-JNDI data sources that you need to get to, you can simply create an implementation of the Hibernate.
 - `org.hibernate.connection.ConnectionProvider` interface.

API Details

- The ConnectionProvider API is fairly straightforward, in consists of four methods:
 - `configure(Properties)` - This method tells the connection provider to prepare itself using the Hibernate properties passed in.
 - `getConnection()` throws `SQLException` - Get a method from your connection source for hibernate to use.
 - `closeConnection(Connection)` throws `SQLException` - Release a connection Hibernate has been using (potentially putting it back in to your legacy pool, or whatever else).
 - `close()` - This method tells the connection provider to release any resources for shutdown.

Hibernate Configuration

- Once implemented, it is simply a matter of telling Hibernate at configuration-time that you want it to use your connection provider:
 - `hibernate.connection.provider_class=com.javalobby.tnt.MyConnectionProvider`

Configuring hibernate here

```
Configuration configuration = new Configuration()  
    .setProperty(Environment.CONNECTION_PROVIDER,  
    "com.javalobby.tnt.MyConnectionProvider");  
  
// providing reference to connection pool here.  
LegacyConnectionPool legacyPool = getLegacyConnectionPool();  
com.javalobby.tnt.MyConnectionProvider.connectionPool = legacyPool;
```

All slides kept here

Hibernate JDBC Properties

Property name	Purpose
hibernate.connection.driver_class	<i>jdbc driver class</i>
hibernate.connection.url	<i>jdbc URL</i>
hibernate.connection.username	<i>database user</i>
hibernate.connection.password	<i>user password</i>
hibernate.connection.pool_size	<i>maximum number of pooled connections</i>

Hibernate Datasource Properties

SQL Dialects

Dialects Translate the mapping details to native SQL statements

Hibernate Configuration Properties

Hibernate JDBC and Connection Properties

Hibernate Cache Properties

Hibernate Transaction Properties
