

## CS520 Web Programming

Object-Relational Mapping with Hibernate

Chengyu Sun  
California State University, Los Angeles

## The Object-Oriented Paradigm

- ◆ The world consists of objects
- ◆ So we use object-oriented languages to write applications
- ◆ We want to store some of the application objects (a.k.a. persistent objects)
- ◆ So we use a Object Database?

## The Reality of DBMS

- ◆ Relational DBMS are still predominant
  - Best performance
  - Most reliable
  - Widest support
- ◆ Bridge between OO applications and relational databases
  - CLI and embedded SQL
  - Object-Relational Mapping (ORM) tools

## Call-Level Interface (CLI)

- ◆ Application interacts with database through functions calls

```
String sql = "select name from items where id = 1";  
  
Connection c = DriverManager.getConnection( url );  
Statement stmt = c.createStatement();  
ResultSet rs = stmt.executeQuery( sql );  
  
if( rs.next() ) System.out.println( rs.getString("name") );
```

## Embedded SQL

- ◆ SQL statements are embedded in host language

```
String name;  
#sql {select name into :name from items where id = 1};  
System.out.println( name );
```

## Employee – Application Object

```
public class Employee {  
  
    Integer id;  
    String name;  
    Employee supervisor;  
  
}
```

## Employee – Database Table

```
create table employees (  
    id            integer primary key,  
    name         varchar(255),  
    supervisor   integer references employees(id)  
);
```

## From Database to Application

- ◆ So how do we construct an Employee object based on the data from the database?

```
public class Employee {  
    Integer      id;  
    String       name;  
    Employee     supervisor;  
  
    public Employee( Integer id )  
    {  
        // access database to get name and supervisor  
        ... ..  
    }  
}
```

## Problems with CLI and Embedded SQL ...

- ◆ SQL statements are hard-coded in applications

```
public Employee( Integer id ) {  
    ...  
    PreparedStatement p;  
    p = connection.prepareStatement(  
        "select * from employees where id = ?"  
    );  
    ...  
}
```

## ... Problems with CLI and Embedded SQL ...

- ◆ Tedious translation between application objects and database tables

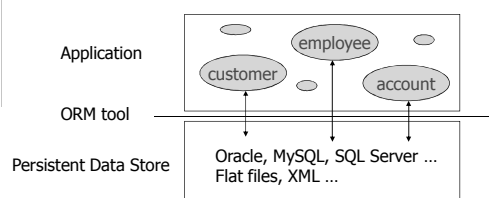
```
public Employee( Integer id ) {  
    ...  
    ResultSet rs = p.executeQuery();  
    if( rs.next() )  
    {  
        name = rs.getString("name");  
        ...  
    }  
}
```

## ... Problems with CLI and Embedded SQL

- ◆ Application design has to work around the limitations of relational DBMS

```
public Employee( Integer id ) {  
    ...  
    ResultSet rs = p.executeQuery();  
    if( rs.next() )  
    {  
        ...  
        supervisor = ??  
    }  
}
```

## The ORM Approach



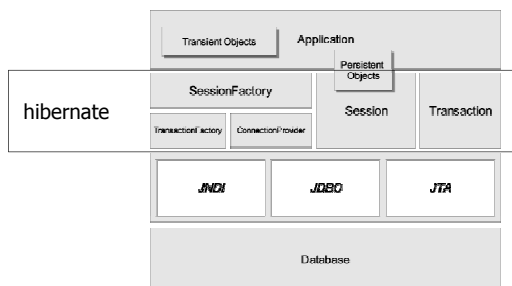
## Advantages of ORM

- ◆ Make RDBMS look like ODBMS
- ◆ Data are accessed as objects, not rows and columns
- ◆ Simplify many common operations. E.g. `System.out.println(e.supervisor.name)`
- ◆ Improve portability
  - Use an object-oriented query language (OQL)
  - Separate DB specific SQL statements from application code
- ◆ Caching

## Common ORM Tools

- ◆ Java Data Object (JDO)
  - One of the Java specifications
  - Flexible persistence options: RDBMS, OODBMS, files etc.
- ◆ Hibernate
  - Most popular Java ORM tool right now
  - Persistence by RDBMS only
- ◆ Java Persistence API (JPA)
  - A *unifying* API standard for Java object persistence
  - Object to *relational* mapping
- ◆ Others
  - [http://en.wikipedia.org/wiki/List\\_of\\_object-relational\\_mapping\\_software](http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software)

## Hibernate Application Architecture



## A Simple Hibernate Application

- ◆ Java classes
  - `Employee.java`
- ◆ O/R Mapping files
  - `Employee.hbm.xml`
- ◆ Hibernate configuration file
  - `hibernate.cfg.xml`
- ◆ (Optional) Logging configuration files
  - `log4j.properties`
- ◆ Code to access the persistent objects
  - `EmployeeTest1.java`
  - `EmployeeTest2.java` (CRUD Example)

## Java Classes

- ◆ Plain Java classes (POJOs); however, it is *recommended* that
  - Each persistent class has an identity field
  - Each persistent class implements the `Serializable` interface
  - Each persistent field has a pair of getter and setter, *which don't have to be public*

## O/R Mapping Files

- ◆ Describe how class fields are mapped to table columns
- ◆ Three important types of elements in a mapping file
  - `<id>`
  - `<property>` - when the field is of simple type
  - Association – when the field is of a class type
    - `<one-to-one>`
    - `<many-to-one>`
    - `<one-to-many>`
    - `<many-to-many>`

## Hibernate Configuration Files

- ◆ Tell hibernate about the DBMS and other configuration parameters
- ◆ Either hibernate.properties or hibernate.cfg.xml or both
  - Database information
  - Mapping files
  - show\_sql

## Access Persistent Objects

- ◆ Session
- ◆ Query
- ◆ Transaction
  - A transaction is required for updates
- ◆ <http://docs.jboss.org/hibernate/stable/core/api/org/hibernate/package-summary.html>

## Hibernate Query Language (HQL)

- ◆ A query language that looks like SQL, but for accessing *objects*
- ◆ Automatically translated to DB-specific SQL statements
- ◆ 

```
select e from Employee e
where e.id = :id
```

  - From all the Employee objects, find the one whose id matches the given value

## More HQL Examples

- ◆ CSNS DAO Implementation classes, e.g.
  - UserDaoImpl.java
  - QuarterDaoImpl.java
- ◆ HQL Features
  - DISTINCT
  - ORDER BY
  - Functions

## Join in HQL ...

```
class User {
    Integer id;
    String username;
    ...
}

class Section {
    Integer id;
    User instructor;
    ...
}
```

users

id	username

sections

id	instructor_id

## ... Join in HQL ...

- ◆ Query: find all the sections taught by the user "cysun".
  - SQL??
  - HQL??

## ... Join in HQL ...

```
class User {
    Integer id;
    String username;
    ...
}

class Section {
    Integer id;
    Set<User> instructors;
    ...
}
```

◆ Database tables??

## ... Join in HQL

- ◆ Query: find all the sections for which "cysun" is one of the instructors
  - SQL??
  - HQL??

## Hibernate Mapping

- ◆ Basic mapping
  - <id>
  - <property>
  - Association
    - many-to-one
- ◆ Advanced mapping
  - Components
  - Collections
  - Subclasses

## hbm2ddl

- ◆ Generate DDL statements from Java classes and mapping files
- ◆ db/hibernate-examples.ddl – generated by hbm2ddl

## Components

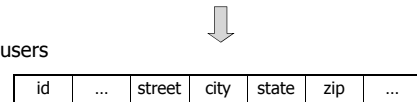
```
public class Address {
    String street, city, state, zip;
}

public class User {
    Integer id;
    String username, password;
    Address address;
}
```

## Mapping Components

```
<component name="address" class="Address">
  <property name="street"/>
  <property name="city"/>
  <property name="state"/>
  <property name="zip"/>
</component>
```

users



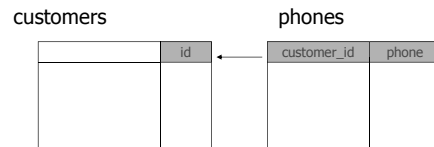
id	...	street	city	state	zip	...
----	-----	--------	------	-------	-----	-----

## Collection of Simple Types

```
public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
}
```

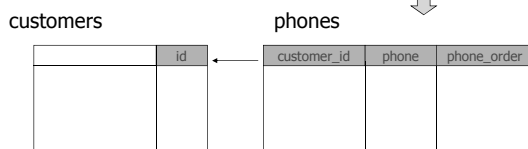
## Set of Simple Types

```
<set name="phones" table="phones">
  <key column="customer_id"/>
  <element type="string" column="phone"/>
</set>
```



## List of Simple Types

```
<list name="phones" table="phones">
  <key column="customer_id"/>
  <index column="phone_order"/>
  <element type="string" column="phone"/>
</list>
```



## Collection of Object Types

```
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
}

public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    Set<Account> accounts;
}
```

## Issues Related to Collections of Object Types

- ◆ Set, List, and Sorted Set
- ◆ Association
  - one-to-many
  - many-to-many
- ◆ Cascading behaviors
- ◆ Unidirectional vs. Bidirectional
- ◆ Lazy loading

## Set of Objects

```
<set name="accounts">
  <key column="customer_id" />
  <one-to-many class="Account" />
</set>
```

*Database tables??*

## List of Objects

```
<list name="accounts">
  <key column="customer_id" />
  <index column="account_order" />
  <one-to-many class="Account" />
</list>
```



*Database tables??*

## Sorted Set of Objects ...

```
<set name="accounts" order-by="created_on asc">
  <key column="customer_id" />
  <one-to-many class="Account" />
</set>
```

- ◆ order-by
- ◆ Objects are sorted in SQL
  - `created_on` is a column, not a property
- ◆ Use `LinkedHashSet` on Java side

## ... Sorted Set of Objects

```
<set name="accounts" sort="natural">
  <key column="customer_id" />
  <one-to-many class="Account" />
</set>
```

- ◆ sort
- ◆ Objects are sorted in Java
- ◆ Use `SortedSet`, e.g. `TreeSet`, on Java side
- ◆ Element class must implements the `Comparable` interface; otherwise a `Comparator` class must be provided

## Cascading Behaviors

```
Customer c = new Customer("cysun");
Account a1 = new Account();
Account a2 = new Account();
c.getAccounts().add( a1 );
c.getAccounts().add( a2 );
```

```
session.saveOrUpdate(c); // will a1 and a2 be saved as well?
```

```
c.getAccounts().remove(a1);
session.saveOrUpdate(c); // will a1 be deleted from db??
```

```
session.delete(c); // will a1/a2 be deleted from db??
```

## Cascading Behaviors in Hibernate ...

- ◆ none (default)
- ◆ save-update
- ◆ delete
- ◆ all (save-update + delete)
- ◆ delete-orphan
- ◆ all-delete-orphan (all + delete-orphan)

## ... Cascading Behaviors in Hibernate

	Save a1&a2	Delete a1	Delete a1/a2
none	N	N	N
save-update	Y	N	N
delete	N	Y	N
all	Y	Y	N
delete-orphan	N	N	Y
all-delete-orphan	Y	Y	Y

## Bidirectional Association – OO Design #1

```
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    Customer owner;
}

public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    Set<Account> accounts;
}
```

## Unidirectional Association – OO Design #2

```
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
}

public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
    Set<Account> accounts;
}
```

## Unidirectional Association – OO Design #3

```
public class Account {
    Integer id;
    Double balance;
    Date createdOn;
    Customer owner;
}

public class Customer {
    Integer id;
    String name;
    String address;
    Set<String> phones;
}
```

## Unidirectional vs. Bidirectional

- ◆ Do the three OO designs result in different database schemas??
- ◆ Does it make any difference on the application side??
- ◆ Which one is the best??

## Mapping Bidirectional Associations

```
<class name="Customer" table="customers">
    ...
    <set name="accounts" inverse="true">
        <key column="customer_id" />
        <one-to-many class="Account" />
    </set>
</class>

<class name="Account" table="accounts">
    ...
    <many-to-one class="Customer" column="customer_id" />
</class>
```

## Lazy Loading

- ◆ Collections are not loaded until they are used
- ◆ But sometimes we want to be "eager"
  - Performance optimization, i.e. reduce the number of query requests
  - Disconnected clients
- ◆ Join fetch

```
from Customers c left join fetch c.accounts
```



## Inheritance

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

## Table Per Concrete Class

accounts

id	balance	created_on
----	---------	------------

cd\_accounts

id	balance	created_on	term
----	---------	------------	------

## Table Per Concrete Class

accounts

id	balance	created_on
----	---------	------------

cd\_accounts

id	balance	created_on	term
----	---------	------------	------

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

## Table Per Subclass

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



cd\_accounts

account_id	term
------------	------

accounts

id	balance	created_on
----	---------	------------

## Table Per Hierarchy

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

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



accounts

id	account_type	balance	created_on	term
----	--------------	---------	------------	------

## O/R Mapping vs. ER-Relational Conversion

O/R Mapping

ER-Relational Conversion

Class ↔ Entity Set

<property> ↔ Attribute

Association ↔ Relationship

Subclass

- table per concrete class
- table per class hierarchy
- table per subclass

Subclass

- OO method
- NULL method
- ER method

## Tips for Hibernate Mapping

- ◆ Understand relational design
  - Know what the database schema should look like before doing the mapping
- ◆ Understand OO design
  - Make sure the application design is object-oriented

## Hibernate Support in Spring

### Without Spring

```
Transaction tx = null;
try
{
    tx = s.beginTransaction();
    s.saveOrUpdate( e );
    tx.commit();
}
catch( Exception e )
{
    if( tx != null ) tx.rollback();
    e.printStackTrace();
}
```

### With Spring

```
getHibernateTemplate()
.saveOrUpdate( user );
```

## Caching in Hibernate

- ◆ Object cache
  - Caching Java objects
  - Simple and effective implementation
    - ◆ Hash objects using identifiers as key
- ◆ Query cache
  - Caching query results
  - No implementation that is both simple and effective

## Cache Scopes

- ◆ Session
- ◆ Process
- ◆ Cluster

## First-Level Cache

- ◆ Session scope
- ◆ Always on (and cannot be turned off)
- ◆ Ensure that there are no duplicate/inconsistent objects in the same session

## Second-Level Cache

- ◆ Pluggable *Cache Providers*
  - Process cache
    - ◆ E.g. EHCACHE, OSCACHE
  - Cluster cache
    - ◆ E.g. SwarmCache, JBossCache
- ◆ Distinguished by
  - Cache scope
  - Concurrency policies

## Isolation Example ...

Sells	bar	beer	price
	Joe's	Bud	2.50
	Joe's	Miller	2.75
	Sue's	Bud	2.50
	Sue's	Miller	3.00

- ◆ Sue is querying `Sells` for the highest and lowest price Joe charges.
- ◆ Joe decides to stop selling Bud and Miller, but to sell only Heineken at \$3.50

## ... Isolation Example

Sue's transaction:

```
-- MAX
SELECT MAX(price) FROM Sells WHERE bar='Joe's';
-- MIN
SELECT MIN(price) FROM Sells WHERE bar='Joe's';
COMMIT;
```

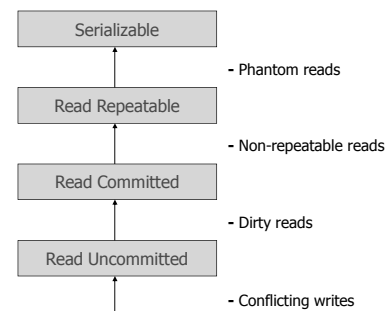
Joe's transaction:

```
-- DEL
DELETE FROM Sells WHERE bar='Joe's';
-- INS
INSERT INTO Sells VALUES( 'Joe's', 'Heineken', 3.50 );
COMMIT;
```

## Potential Problems of Concurrent Transactions

- ◆ Caused by *interleaving operations*
- ◆ Caused by *aborted operations*
- ◆ For example:
  - MAX, DEL, MIN, INS
  - MAX, DEL, INS, MIN

## Transaction Isolation Levels



## Currency Support of Hibernate Cache Providers

	Read-only	Nonstrict Read-Write	Read-Write	Transactional
EHCACHE	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBossCache	X			X

Read Uncommitted: Read-only, Nonstrict Read-Write, Read-Write  
 Read Committed: Read-only, Nonstrict Read-Write, Read-Write, Transactional  
 Read Repeatable: Read-only, Nonstrict Read-Write, Read-Write, Transactional

## Readings

- ◆ *Java Persistence with Hibernate* by Christian Bauer and Gavin King (or *Hibernate in Action* by the same authors)
- ◆ Hibernate Core reference at <http://docs.jboss.org/hibernate/stable/core/reference/en/html/>
  - Chapter 5-10, 15

## More Readings

- ◆ *Database Systems – The Complete Book* by Garcia-Molina, Ullman, and Widom
  - Chapter 2: ER Model
  - Chapter 3.2-3.3: ER to Relational Conversion
  - Chapter 4.1-4.4: OO Concepts in Databases
  - Chapter 9: OQL
  - Chapter 8.7: Transactions