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

```java
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

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

Employee – Application Object

```java
public class Employee {
    Integer id;
    String name;
    Employee supervisor;
}
```
Employee – Database Table

```sql
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?

```java
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

```java
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

```java
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

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

The ORM Approach

```
Application
ORM tool
Persistent Data Store
Oracle, MySQL, SQL Server ...
Flat files, XML ...
```

- Employee
- Customer
- Account
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
- Others

Hibernate Application Architecture

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>

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)
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

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 ...

```java
class User {
    Integer id;
    String username;
    ...
}
class Section {
    Integer id;
    User instructor;
    ...
}
```

... 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>

table customers
| id | customer_id | phone |

List of Simple Types

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

table customers
| id | customer_id | phone | phone_order |

Collection of Object Types

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

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

Issues Related to Collections of Object Types

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

Set of Objects

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

database tables??
List of Objects

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

Sorted Set of Objects...

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

... Sorted Set of Objects

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

Cascading Behaviors in Hibernate

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

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

```java
from Customers c left join fetch c.accounts
```
Bidirectional Association – OO Design #1

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

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

Unidirectional Association – OO Design #2

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

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

Unidirectional Association – OO Design #3

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

```java
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

```xml
<class name="Customer" table="customers">
  ...  
</class>

<class name="Account" table="accounts">
  ...  
</class>
```

Inheritance

```java
public class CDAccount extends Account {
    Integer term;
}
```
Table Per Concrete Class

accounts

<table>
<thead>
<tr>
<th>id</th>
<th>balance</th>
<th>created_on</th>
</tr>
</thead>
</table>

cd_accounts

<table>
<thead>
<tr>
<th>id</th>
<th>balance</th>
<th>created_on</th>
<th>term</th>
</tr>
</thead>
</table>

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

Table Per Subclass

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

cd_accounts

<table>
<thead>
<tr>
<th>id</th>
<th>balance</th>
<th>created_on</th>
<th>term</th>
</tr>
</thead>
</table>

accounts

<table>
<thead>
<tr>
<th>id</th>
<th>balance</th>
<th>created_on</th>
</tr>
</thead>
</table>

Table Per Hierarchy

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

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

accounts

<table>
<thead>
<tr>
<th>id</th>
<th>account_type</th>
<th>balance</th>
<th>created_on</th>
<th>term</th>
</tr>
</thead>
</table>

O/R Mapping vs. ER-Relational Conversion

<table>
<thead>
<tr>
<th>O/R Mapping</th>
<th>ER-Relational Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Entity Set</td>
</tr>
<tr>
<td>&lt;property&gt;</td>
<td>Attribute</td>
</tr>
<tr>
<td>Association</td>
<td>Relationship</td>
</tr>
<tr>
<td>subclass</td>
<td></td>
</tr>
<tr>
<td>• table per concrete class</td>
<td>• OO method</td>
</tr>
<tr>
<td>• table per class hierarchy</td>
<td>• NULL method</td>
</tr>
<tr>
<td>• table per subclass</td>
<td>• ER method</td>
</tr>
</tbody>
</table>

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

```java
Without Spring
Transaction tx = null;
try {
    tx = s.beginTransaction();
s.saveOrUpdate( user );
    tx.commit();
} catch ( Exception e )
    { if ( tx != null ) tx.rollback();
        e.printStackTrace();
    }
```

```java
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 ...

<table>
<thead>
<tr>
<th>Sells</th>
<th>bar</th>
<th>beer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe’s</td>
<td>Bud</td>
<td>2.50</td>
<td></td>
</tr>
<tr>
<td>Joe’s</td>
<td>Miller</td>
<td>2.75</td>
<td></td>
</tr>
<tr>
<td>Sue’s</td>
<td>Bud</td>
<td>2.50</td>
<td></td>
</tr>
<tr>
<td>Sue’s</td>
<td>Miller</td>
<td>3.00</td>
<td></td>
</tr>
</tbody>
</table>

- 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
### Potential Problems of Concurrent Transactions

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

### Transaction Isolation Levels

- **Serializable**
  - Phantom reads
- **Read Repeatable**
  - Non-repeatable reads
- **Read Committed**
  - Dirty reads
- **Read Uncommitted**
  - Conflicting writes

### Currency Support of Hibernate Cache Providers

<table>
<thead>
<tr>
<th>Cache</th>
<th>Read-only</th>
<th>Non-strict Read-Write</th>
<th>Read-Write</th>
<th>Transactional</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHCache</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OSCache</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SwarmCache</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>JBossCache</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### Readings

- **Java Persistence with Hibernate** by Christian Bauer and Gavin King (or **Hibernate in Action** by the same authors)
- **Hibernate Core reference at** http://www.hibernate.org
  - Chapter 3-10, 14

### 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: QL
  - Chapter 8.7: Transactions