Mapping Objects to Relational Databases using Hibernate

German Eichberger

About me

German Eichberger, M.S., is a senior software engineer with UCSD's fMRI Center, lecturer and former program manager with UCSD extension, an adjunct professor with Mesa College, and the founder and CEO of e-nnovate Technologies Inc. Prior to that he worked as a project manager and technical architect for PricewaterhouseCoopers designing and implementing e-commerce, document management and CRM solutions. He earned his degree in computer vision research and vehicle tracking from the University of Karlsruhe.

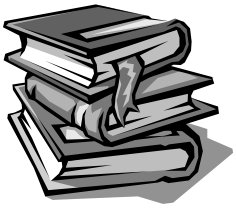
Agenda

- Introduction
- Basic Mapping Concepts
- Hibernate Lab 1
- Mapping Inheritance Structures
- Hibernate Lab 2
- Lunch Break
- Mapping Object Relationships
- Hibernate Lab 3
- Performance Tuning
- Conclusion

Introduction

- What is persistence?
- The paradigm mismatch
- Persistence Layers and alternatives
- What is ORM?
- The Role of the Agile DBA

What is persistence?



From Wikipedia, the free encyclopedia:

Persistence is the term used in computer science to describe a capability used by a computer programmer to store data structures in non-volatile storage such as a file system or a **relational database**.

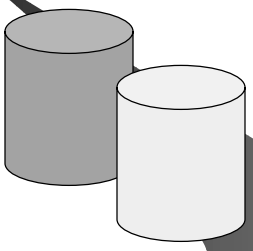
Without this capability data structures only exist in-memory, and will be lost when a program exits. Persistence allows, for example, a program to be restarted and re-loaded with the data structures from a previous invocation of the program.

Design patterns solving this problem are **container based** persistence, component based persistence and the Data Access Object model.

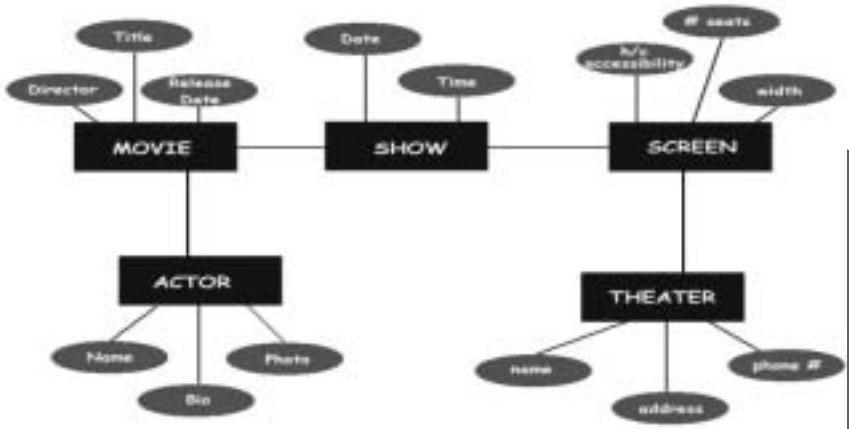
Examples of persistence are using Java **serialization** to store Java objects on disk or using J2EE to store Enterprise Java Beans in a relational database.

Relational Databases

- present a view of data as a collection of rows and columns
- Codd's 12 Rules
- Used almost everywhere
- Common denominator in most corporations
- SQL
- **E**ntity **R**elationship Diagram



E-R Diagram



SQL

- `Create` and `Alter`
- `Select, Insert, Delete`
- Joins and Cartesian Product
- Group, order
- A sound knowledge of SQL is mandatory for sound Java database application

SQL in Java, C/C++, …

- Usually language in language
- Special “connectors” to the database, e.g. JDBC, ADO.NET, ODBC, …
- RDMS dominate the computing industry
- SQL is the language of choice

Persistence in OOP

- Difference between persistent and transient objects
- Business logic doesn’t interact with rows and columns in DB
- Business logic interacts within object-oriented domain model
- Therefore it is possible to use sophisticated object-oriented objects

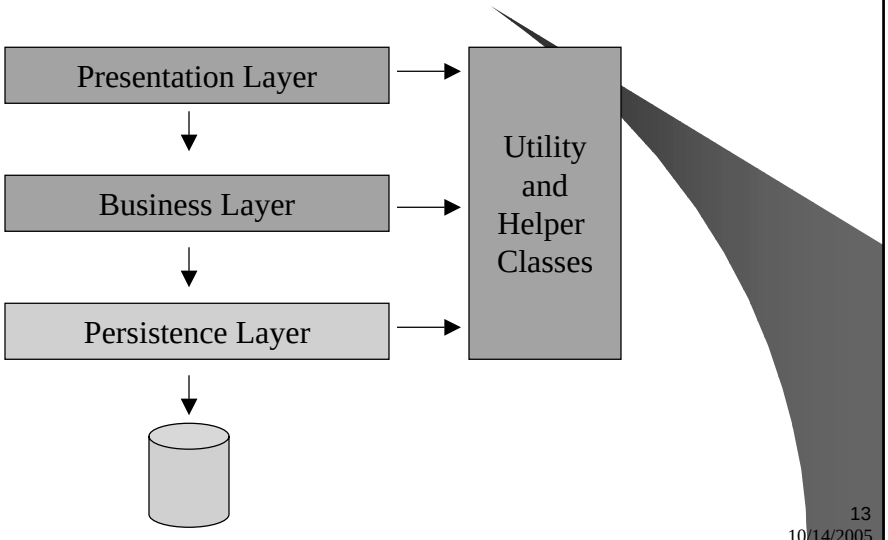
The paradigm mismatch (1)

- Granularity
  - OOP: coarse-grained and finer-grained classes, simple datatypes (e.g. String)
  - RDMBS: Tables and columns
- Inheritance and Polymorphism
- Identity
  - OOP: a==b a.equals(b)
  - RDBMS: primary keys are the same

The paradigm mismatch (2)

- Associations
  - OOP: Object references, one-to-many, many-to-many,…
  - RDMBS: joins, link table, …
- Object graph navigation
  - OOP: a.getB().getBValue()
  - RDBMS: select Bvalue from a,b where a.b_id = b.id and a.id = 123
- The cost
  - 30% of application code written deals with the mismatch

Layered Architecture



Implement a Persistence Layer

- Hand-Code with SQL/JDBC or similar
  - Different SQL Dialects
  - Lots of work
  - Hard to test
  - “Not invented here”
- Serialization
  - Can only access all objects as once
  - Not very good for concurrent enterprise applications

Implement a Persistence Layer

- EJB 2.0 entity beans
  - Bean Managed Persistence
  - Container managed persitence
  - Unpopular -- so being abandoned and replaced by JDO (EJB 3.0)
  - Why EJB's are bad
    - Too coarse grained - 1:1 relationship with tables
    - Too fine grained for reusability
    - No polymorphic associations and queries
    - Not portable between different application servers
    - Not serializable
    - Forces an unatural Java Style

Implement a Persistence Layer

- Object Oriented Database Systems
  - Not very popular
  - But with JDO there might be a come-back
  - According to vendors much faster than an RDBMS
  - And who is still dealing with ISAM?
- Others
  - XML persistence
  - …

What is ORM?

"The automated and transparent persistence of objects to the tables in a relational database, using metadata that describes the mapping between the objects and the database"

Parts of an ORM solution

- An API for performing basic operations
- An API or language for queries
- A way to specify the mapping metadata
- Some tricks like
  - Dirty checking
  - Lazy association fetching
  - Automatic optimization

Ways to implement 1

- Pure relational
  - The application is designed around tables
  - Heavy use of “stored procedure”
- Light object mapping
  - Classes are mapped manually to tables
  - Design Patterns hide SQL from Objects

Ways to implement 2

- Medium Object Mapping
  - The Objects are the center
  - SQL is generated at build time
  - Objects are **cached** by the persistence layer
- Full object mapping
  - Sophisticated object modeling
  - Transparent persistency
  - Efficient fetching and caching strategies

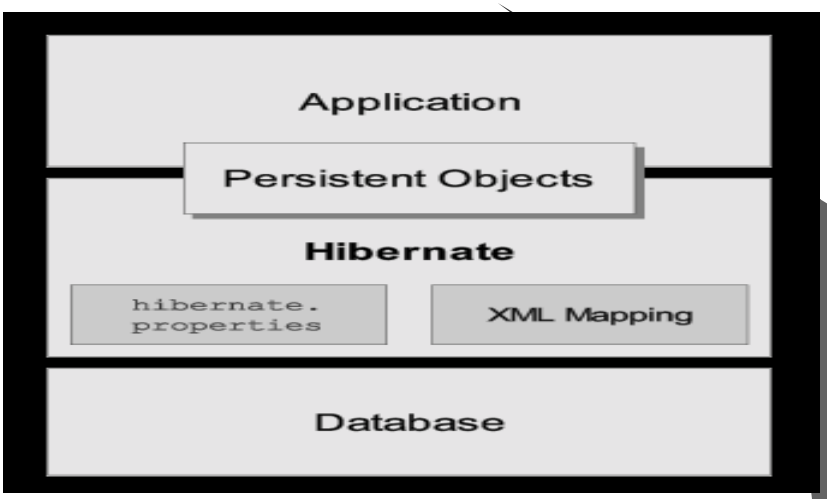
Why ORM

- Productivity
- Maintainability
- Performance
- Vendor Independence

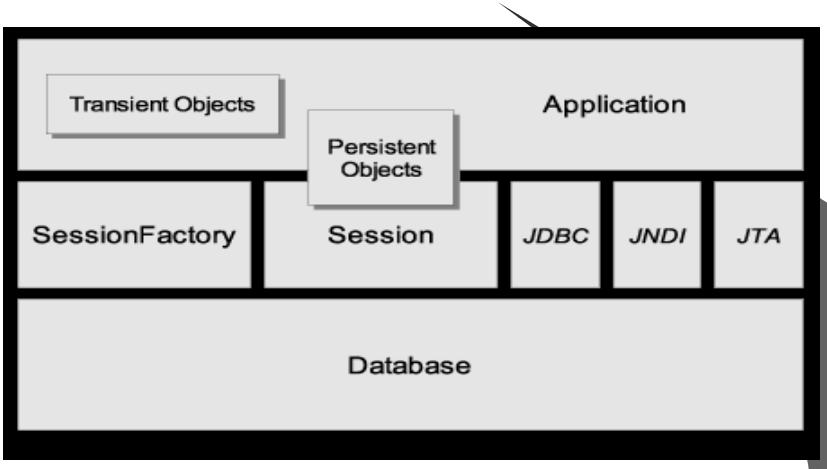
Hibernate is an Open Source ORM Technology

- Compatible with EJB 3.0
- Transparent Persistence
- Flexible Mapping
- Query Facilities
- Metadata Facilities
- Hibernate Multi-Layer Cache Architecture
- Performance
- Extension Points

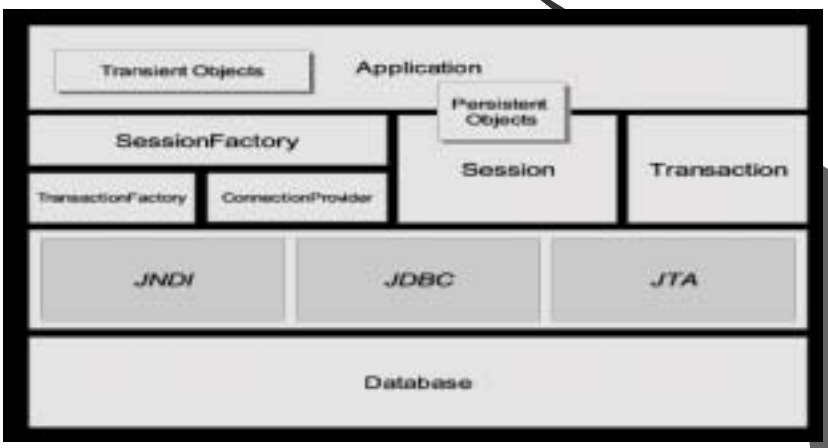
Architecture



Architecture - Lite



Architecture – Full Cream



Basic Mapping Concepts

Basic Mapping Concepts

- Introduction
- Mapping Terminology
- Shadow Information
- Mapping Meta data

Introduction

- Start with the data attributes of a **class**
- An attribute will map to zero or more columns in the database
- Not all attributes are persistent
- Some attributes are objects themselves
  - Really reflects an association
  - The object needs to be mapped as well

Introduction 2

- Mapping is recursive and will end
- Easiest mapping is a one attribute to a single column
- Usually classes don't map to single tables
  - But it's a good start
  - Performance tuning might motivate **Refactoring**
- In new projects the object schema should drive the data model

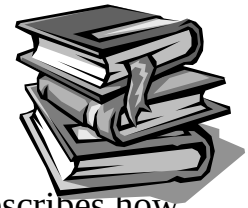
Mapping Terminology

Mapping (v): The act of determining how objects and their relationships are persisted in permanent data storage

Mapping (n): The definition of how an object's property or a relationship is persisted in permanent storage

Property: A data attribute, either implemented as a physical attribute such as the string *firstname* or as a virtual attribute implemented via an operation such as *getTotal()*

Mapping Terminology



Property mapping: A mapping that describes how to persist an object property.

Relationship mapping: A mapping that describes how to persist a relationship (association, aggregation, or composition) between two or more objects.

Shadow information

- Any data an object needs to maintain above and beyond its normal domain data to persist itself
  - Primary key
  - Concurrency control
  - Boolean isPersistent()
- Common Style Convention in UML is to not show shadow information

Mapping Meta Data

- Mapping Table
- Use a picture like on the whiteboard
- Write it in Hibernate's XML language
- Use EJB 3.0 Annotations

Hibernate Lab 1

Hibernate Lab 1

- Hello World
- Understanding the architecture
- Mapping a basic **POJO**

Mapping Inheritance Structures

Mapping Inheritance Structure

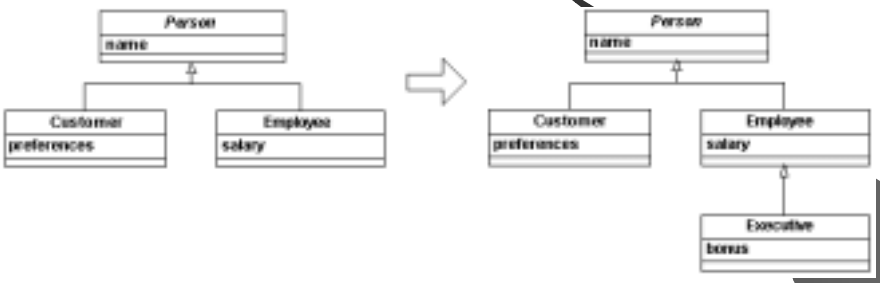
- Introduction
- Map Hierarchy to a single table
- Map each concrete class to its own table
- Map each class to its own table
- Map classes to a generic table structure
- Mapping multiple inheritance

Introduction

- RDMS do not natively support inheritance
- Inheritance results in some interesting twists when saving objects

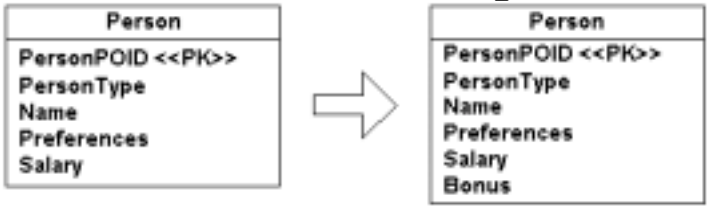
For simplicity the following examples are not tuned.

A simple hierarchy



Note: This is not the best modeling -- if somebody is a customer and an employer things get confusing.

Map Hierarchy to a single table



- One table holds all attributes from multiple hierarchies
- With more hierarchies things get messy
- Imagine: Executives who are customers…

- Refactor

Hibernate Mapping

```
<class name="Person" table="Person">
   <id name="PersonPOID" type="long" column="PersonPOID">
      <generator class="native"/>
   </id>
   <discriminator column="PERSON_TYPE" type="string"/>
   <property name="Name" column="Name"/>
   <subclass name="Customer" discriminator-value="CUSTOMER">
      <property name="Preferences" column="Preferences"/>
   </subclass>
   <subclass name="Employee" discriminator-value="EMPLOYEE">
      <property name="Salary" column="Salary"/>
     <subclass name="Executive" discriminator-value="EXECUTIVE">
           <property name="Bonus" column="Bonus"/>
     </subclass>
   </subclass>
</class>
```


Hibernate Mapping

```
@Entity()
@Inheritance(
    strategy=InheritanceType.SINGLE_TABLE,
    discriminatorType=DiscriminatorType.STRING,
    discriminatorValue="Person"
)
@DiscriminatorColumn(name=" PERSON_TYPE ")
public class Person { ... }

@Entity()
@Inheritance(
    discriminatorValue="CUSTOMER"
)
public class Customer extends Person { ... }
```

Each concrete class to own table

- One table for each class including the
  - Attributes implemented by the class
  - Attributes inherited

Hibernate – Mapping 1

```
<class name="Customer" table="Customer">
   <id name="CustomerPOID" type="long"
   column="CustomerPOID">
         <generator class="native"/>
   </id>
 <property name="Name" column="Name"/>
<property name="Preference"
   column="Preference"/>
</class>
```

Hibernate – Mapping 2

```
<class name="Person">
   <id name="POID" type="long" column="POID">
      <generator class="native"/>
   </id>
   <property name="Name" column="Name"/>
  <union-subclass name="Customer" table="Customer">
      <property name="Preference" column="Preference"/>
 </union-subclass>
   <union-subclass name="Employee" table="Employee">
      ...
            <union-subclass name="Executive" table="Executive">
            ...
            </union-subclass>
   </union-subclass>
</class>
```


Hibernate – Mapping

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Person implements Serializable {
```

Each Class to own table

- Customer and Employer in two tables -> join
- Note: Lots of keys and relations
- **Views**
- Type column in person

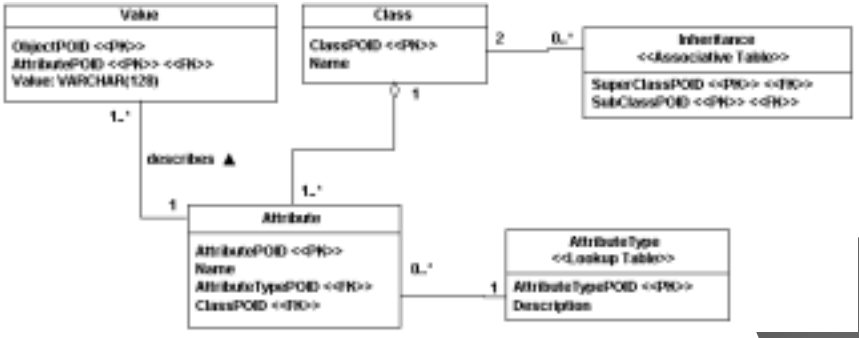
Hibernate Mapping

```
<class name="Person" table="Person">
   <id name="PersonPOID" type="long" column="PersonPOID">
      <generator class="native"/>
   </id>
   <property name="Name" column="Name"/>
  <joined-subclass name="Customer" table="Customer">
      <key column="PersonPOID"/>
      <property name="Preference" column="PREFERENCE"/>
 </joined-subclass>
   <joined-subclass name="Employee" table="Employee">
      <key column="PersonPOID"/>
     …
     <joined-subclass name="Executive" table="Executive">
      <key column="PersonPOID"/>
      ...
   </joined-subclass>
</joined-subclass>
</class>
```

Hibernate Mapping

```
@Entity()
@Inheritance(strategy=InheritanceType.JOINED )
public class Person implements Serializable { ... }

@Entity()
public class Customer extends Person { ... }

@Entity(access=AccessType.FIELD)
 public class Employee extends Person { ... }
```

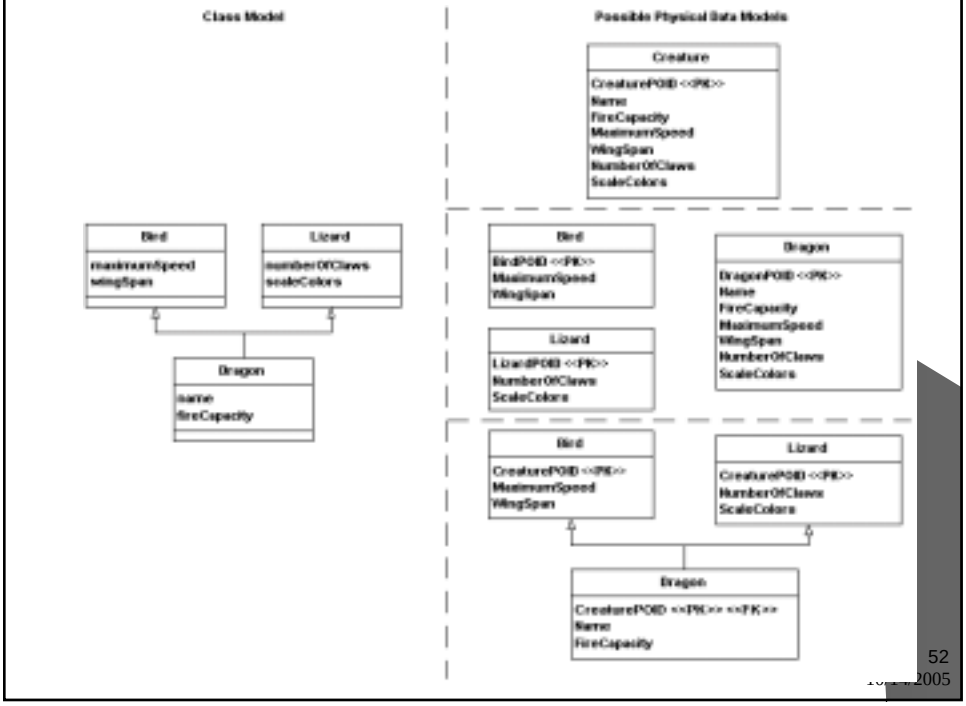

Generic Table Structure



- Very flexible approach
- Used when end users can add their own attributes
- Slow

Multiple inheritance

- Definition: A class inherits from two or more superclasses
- A questionable feature of OO which rarely make sense
  - Not supported by Java



Conclusion

- None of these strategies are ideal for all solutions
- The easiest strategy
  - One table per hierarchy
  - Then refactor
- One table per concrete class is messy if you need to refactor
  - data migration nightmare
- Generic Schema is slow

Hibernate Lab 2

Hibernate Lab 2

- Hibernate generates schema
- Mapping class inheritance
  - Table per concrete class
  - Table per class hierarchy
  - Table per subclass
- Simple Query

Mapping Object Relationships

Mapping Object Relationships

- Types of Relationships
- How Object Relationships are implemented
- How Relational Database Relationships are implemented
- Relationship Mapping
- Mapping Ordered Collections
- Mapping Recursive Relationships
- Mapping Class Scope Properties

Types of Relationships

- First category based on multiplicity
  - One-to-one relationship
  - One-to-many
  - Many-to-many relationship
- Directionality
  - Uni-directional
    - Not supported by RDBMS, hence all relationships are bi-directional
  - Bi-directional

How Object Relationships are implemented

- Combination of reference and operations
  - Multiplicity one: Reference and getter/setter
  - Multiplicity is many: collection Attribute
    - Array, List, HashMap, etc.
  - Uni-directional only implemented in one class
  - Bi-directional implemented in two classes

How Relational Database Relationships are Implemented

- Relationships are maintained through the use of foreign keys
- One-to-one and One-to-many just have the foreign key as column in the other table
- Many-to-Many
  - Either multiple foreign key columns
  - Special Many-to-Many table

Tips on keys

- Use single column keys
- Let the database generate the keys for you
  - Hibernate has its own generators but also can use the ones provided by the DB

Relationship Mappings

- Keep the multiplicities the same
  - You could map a one-to-one relationship as a many-to-many. Why?
- When reading one object Hibernate can automatically read all the other objects defined in the relation into memory

One-To-One Mappings

- Let's work through the logic of retrieving a single Position object one step at a time:
  1. The Position object is read into memory.
  2. The holds relationship is automatically traversed.
  3. The value held by the Position.EmployeePOID column is used to identify the single employee that needs to be read into memory.
  4. The Employee table is searched for a record with that value of EmployeePOID.
  5. The Employee object (if any) is read in and instantiated.
  6. The value of the Employee.position attribute is set to reference the Position object.

One-To-One Mappings 2

- Now let's work through the logic of retrieving a single Employee object one step at a time:
  1. The Employee object is read into memory.
  2. The holds relationship is automatically traversed.
  3. The value held by the Employee.EmployeePOID column is used to identify the single position that needs to be read into memory.
  4. The Position table is searched for a row with that value of EmployeePOID.
  5. The Position object is read in and instantiated.
  6. The value of the Employee.position attribute is set to reference the Position object.

One-To-One Mappings 3

- Let's assume we save an object
  - Create a **Transaction** to maintain **referential integrity**
  - Add update statements for each object
  - Each update statement contains both business attributes and key values mapped
  - Because of the foreign keys being inserted the relationship is persisted

One-To-One Mapping 4

- Look at Employer-Position
  - Foreign key is implemented in position
  - But in the object schema it is in employer
  - It works both ways
    - But a future requirement might be that an employer can have multiple positions
  - Requires the judgment call from a good Mapping Specialist…

Hibernate Mapping

The foreign key with a unique constraint, from Employee to Position, may be expressed as:

```
<many-to-one name=“Position" class=“Position"
    column=“PositionPOID" unique="true"/>
```

And this association may be made bidirectional by adding the following to the Employee mapping:

```
<one-to-one name="employee" class="Employee" property-
    ref=“position"/>
```

Hibernate Mapping

```
@Entity
public class Employee {
   @Id
   public Long getId() { return id; }

   @OneToOne(cascade = CascadeType.ALL)
   @PrimaryKeyJoinColumn
   public Position getPosition() {
      return position;
   }
   ...
}
```


One-to-Many Mappings

- Look at Employee and Division
  - Automatically traversed from employee to division
  - And in Hibernate vice versa
  - Also cascading updates and deletes
    - Performance?

One-to-Many Mappings 2

- Employee read into memory traverses the relationship
- But you don't want to have several copies of the same division
- Hibernate needs to keep track of that
  - Caching
- Adding an employee to a division is just `divsision.addEmployee(employee)`

One-to-Many Mappings 3

- Saving works the same way as in One-To-One
- Note: Every example uses a generic key as foreign key but you could also use an attribute which is unique, e.g. the social security number
  - I prefer generic keys though

Hibernate Mapping

```
<many-to-one name="employee" class="Division"
   column="Division_POID"/>
```

Reverse

```
<bag
  name="employees"
  <key column="Division_POID"/>
 <one-to-many class="Employee"/>
</bag>
```


Hibernate Mapping

```
@Entity()
public class Employee implements Serializable {
   @ManyToOne( cascade =
   {CascadeType.CREATE,
   CascadeType.MERGE} )
   @JoinColumn(name="DivisionPOID")
   public  getDivision() {
      return division;
   }
   ...
}
```

Many-to-Many Mappings

- **Associative Table**, a data entity whose sole purpose is to maintain the relationship between two or more tables in the database
- Look at Employee and Task
  - Additional Table
  - Usually contains the keys
- Notice that the multiplicity still works out

Many-to-Many Mappings 2

- Assume we need to retrieve all tasks for an employee object in memory
  1. Create a SQL Select statement that joins the EmployeeTask and Task tables together, choosing all EmployeeTask records with the an EmployeePOID value the same as the employee we are putting the task list together.
  2. The Select statement is run against the database.
  3. The data records representing these tasks are marshaled into Task objects. Part of this effort includes checking to see if the Task object is already in memory. If it is then we may choose to refresh the object with the new data values (this is a concurrency issue).
  4. The Employee.addTask() operation is invoked for each Task object to build the collection up.

Many-to-Many Mappings 3

- Assume we need to save the relationship
  1. Start a transaction.
  2. Add Update statements for any task objects that have changed.
  3. Add Insert statements for the Task table for any new tasks that you have created.
  4. Add Insert statements for the EmployeeTask table for the new tasks.
  5. Add Delete statements for the Task table any tasks that have been deleted. This may not be necessary if the individual object deletions have already occurred.
  6. Add Delete statements for the EmployeeTask table for any tasks that have been deleted, a step that may not be needed if the individual deletions have already occurred.
  7. Add Delete statements for the EmployeeTask table for any tasks that are no longer assigned to the employee.
  8. Run the transaction.

Many-to-Many Mappings 4

- Many-To-Many Realtionships map two business classes to three tables

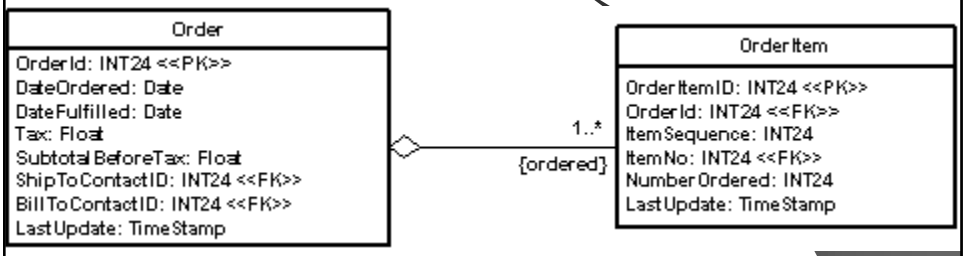
Hibernate Mapping

```
<bag
   name="tasks"
  table="EMPLOYEE_TASKS" >
   <key column="Employee_POID"/>
  <many-to-many
      class="Task"
      column="Task_POID"
   />
</bag>
```


Hibernate Mapping

```
@Entity
public class Employer implements Serializable {
   @ManyToMany(
      targetEntity=org.hibernate.test.metadata.manytomany.Employee.class,
      cascade={CascadeType.CREATE, CascadeType.MERGE}
   )
   @JoinTable(
      table=@Table(name="EMPLOYER_TASKS"),
      joinColumns={@JoinColumn(name="Employee_POID")},
      inverseJoinColumns={@JoinColumn(name="Task_POID")}
   )
   public Collection getTasks() {
      return tasks;
   }
   ...
}
```

Mapping Ordered Collections



- Ordered constrained placed on the relationship
- Additional column ItemSequence

Mapping Ordered Collections

- Read the data in the proper sequence
- Don’t include the sequence number in the key
- When to update sequence numbers after rearranging order items
- Update sequence numbers after deleting an order item
- Consider sequence gaps greater than one

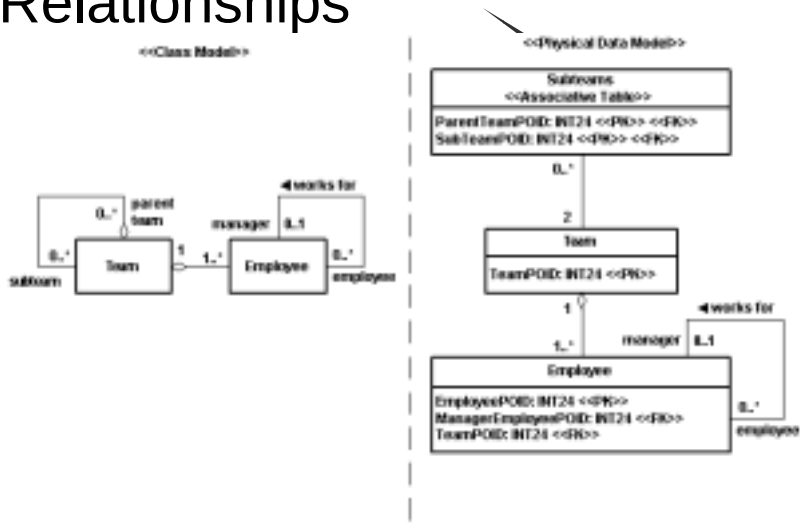
Hibernate Mapping (in memory)

```
<set name=“OrderItem"
        table=“OrderItem"
        sort="natural">
   <key column=“OrderId"/>
   <element column=“ItemSequence" type=“long"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
   <key column="year_id"/>
   <map-key column="hol_name" type="string"/>
   <element column="hol_date" type="date"/>
</map>
```


Hibernate Mapping (in DB)

```
<set name=“OrderItem" table=“OrderItem" order-by="
   ItemSequence asc">
   <key column=“OrderId"/>
   <element column=“ItemSequence" type=“long"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
   <key column="year_id"/>
   <map-key column="hol_name" type="string"/>
   <element column="hol_date type="date"/>
</map>
```

Mapping Recursive Relationships

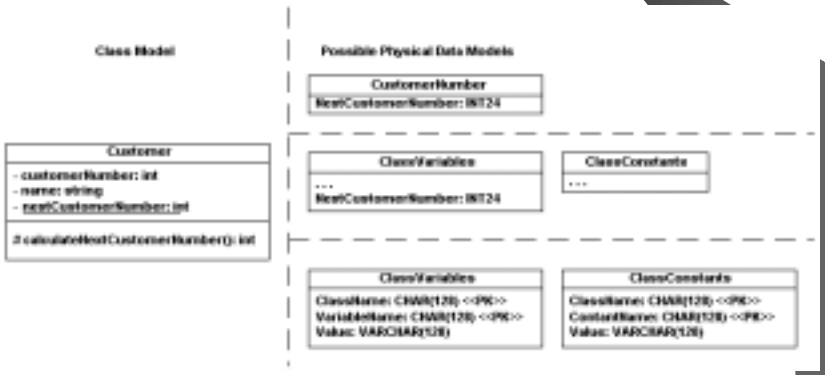


Mapping Recursive Relationships

- Also called *reflexive relationship*
- Mapped the same way as many-to-many relationships
  - In the **associate table** both columns are foreign keys into the same table

Mapping Class Scope Properties

- Basically static variables
  - Available to all instances not just one



Hibernate Lab 3

Hibernate Lab 3

- Implement Relations
  - One-to-One
  - One-to-Many
  - Many-to-Many

Hibernate Query Language

Hibernate Query Language

Why to use HQL?

- Full support for relational operations:
  - HQL allows representing SQL queries in the form of objects.
  - Hibernate Query Language uses Classes and properties instead of tables and columns.
- Return result as Object:
  - The HQL queries return the query result(s) in the form of object(s), which is easy to use.
  - This eleminates the need of creating the object and populate the data from result set.

Why to use HQL?

- Polymorphic Queries:
  - HQL fully supports polymorphic queries.
  - Polymorphic queries gives the query results along with all the child objects if any.
- Easy to Learn:
  - Hibernate Queries are easy to learn and can be easily implemented in applications.
- Support for Advance features:
  - HQL contains many advanced features such as pagination, fetch join with dynamic profiling, Inner/outer/full joins, Cartesian products.
  - HQL also supports Projection, Aggregation (max, avg) and grouping, Ordering, Sub queries and SQL function calls.

Why to use HQL?

- Database independent:
  - Queries written in HQL are database independent

Sample Test program

```
public class SelectHQLExample {
  public static void main(String[] args) {
  Session session = null;
  try{
    // This step will read hibernate.cfg.xml and prepare
   hibernate for use
    SessionFactory sessionFactory = new
   Configuration().configure().buildSessionFactory();
     session =sessionFactory.openSession();
         String SQL_QUERY =“Put your HQL query here";
       Query query = session.createQuery(SQL_QUERY);
       for(Iterator it=query.iterate();it.hasNext();){
         …
       }
          session.close();
  }catch(Exception e){
    System.out.println(e.getMessage());
  }finally{
    }
  }
}
```


From

- `from Cat as cat`
- cartesian product or "cross" join.
  - `from Formula as form, Parameter as param`

Associations and joins

- The supported join types are borrowed from ANSI SQL
  - **inner join**
  - left outer join
  - right outer join
  - full join (not usually useful)

A left outer join returns all the records from the left table, or the one side of a relationship.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as
  kitten
```


The select clause

- The select clause picks which objects and properties to return in the query result set.
  - `select cat.mate from Cat cat`
  - `select cat.name from DomesticCat cat where cat.name like 'fri%'`

The select clause 2

- Queries may return multiple objects and/or properties as an array of type Object[],
  - `select new list(mother, offspr, mate.name) from DomesticCat as mother inner join mother.mate as mate left outer join mother.kittens as offspr`
  - `select new Family(mother, mate, offspr) from DomesticCat as mother join mother.mate as mate left join mother.kittens as offspr`

Aggregate functions

- The supported aggregate functions are
  - avg(...), sum(...), min(...), max(...)
  - count(*)
  - count(...), count(distinct ...), count(all...)

```
select avg(cat.weight),
  sum(cat.weight), max(cat.weight),
  count(cat)
from Cat cat
```

Aggregate functions 2

- ```
  select cat.weight +
  sum(kitten.weight)
  from Cat cat join cat.kittens
  kitten group by cat.id,
  cat.weight
  ```

- ```
  select firstName||' '||initial||'
  '||upper(lastName) from Person
  ```


Polymorphic queries

- from Cat as cat
- All persistent Objects
  - from java.lang.Object o

Where clause

- select foo from Foo foo, Bar bar where foo.startDate = bar.date
- from bank.Person person where person.id.country = 'AU' and person.id.medicareNumber = 123456
- from Cat cat where cat.class = DomesticCat

Expressions

```
select cust
from Product prod, Store store
  inner join store.customers cust
where prod.name = 'widget'
  and store.location.name in (
  'Melbourne', 'Sydney' )
  and prod = all
  elements(cust.currentOrder.lineIt
  ems)
```

Order By, Group By

- from DomesticCat cat order by cat.name asc, cat.weight desc, cat.birthdate
- select cat.color, sum(cat.weight), count(cat) from Cat cat group by cat.color
- select cat.color, sum(cat.weight), count(cat) from Cat cat group by cat.color having cat.color in (eg.Color.TABBY, eg.Color.BLACK)

Example

- The following query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value.
- In determining the prices, it uses the current catalog.
- The resulting SQL query, against the ORDER, ORDER_LINE, PRODUCT, CATALOG and PRICE tables has four inner joins and an (uncorrelated) subselect.

Example

- select order.id, sum(price.amount), count(item)
- from Order as order join order.lineItems as item join item.product as product, Catalog as catalog join catalog.prices as price
- where order.paid = false
  and order.customer = :customer
  and price.product = product
  and catalog.effectiveDate < sysdate
  and catalog.effectiveDate >= all (
  select cat.effectiveDate
  from Catalog as cat
  where cat.effectiveDate < sysdate)
  group by order
  having sum(price.amount) > :minAmount
  order by sum(price.amount) desc

Hibernate Lab 4

Hibernate Lab 4

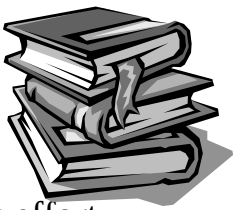
- Generate Table
- Generate Pojo
- Generate Queries
  - From
  - Select
  - Where
  - Order by
  - Group by

Performance Tuning

Performance Tuning

- Definitions
- Tuning Mappings
- Lazy Reads
- Caching

Definitions



- **Database performance tuning:** This effort focuses on changing the database schema itself, often by denormalizing portions of it. Other techniques include changing the types of key columns, for example an index is typically more effective when it is based on numeric columns instead of character columns; reducing the number of columns that make up a composite key; or introducing indices on a table to support common joins.

Definitions

- **Data access performance tuning.** This effort focuses on improving the way that data is accessed. Common techniques include the introduction of stored procedures to "crunch" data in the database server to reduce the result set transmitted across the network; reworking SQL queries to reflect database features; clustering data to reflect common access needs; and caching data within your application to reduce the number of accesses.

Tuning Mappings

- More than one way to map an object
  - Four ways for inheritance
  - Two ways for one-to-one relationships
  - Four ways for class scope properties
- Play around to see what's best
- Note: Whenever you change a mapping strategy you also have to change the object schema and/or the data schema

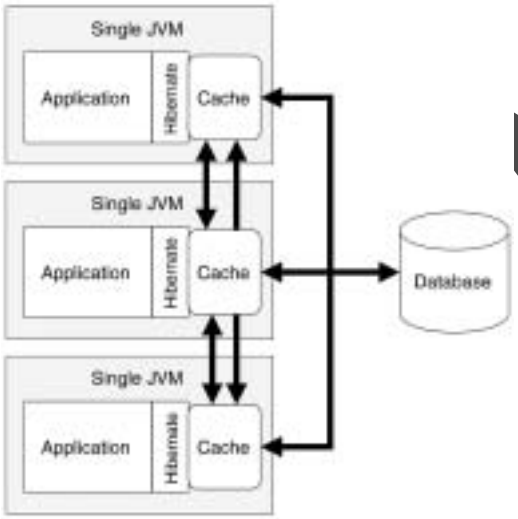
Lazy Reads

- Should attributes be automatically read in when an object is retrieved?
- Some attributes are rather large and rarely accessed
- Lazy means: Instead of loading the attribute when the object is retrieved it is read when the attribute is accessed
- Lazy Read is commonly used in searching and (drill-down) reporting

Caching

- Use a connection Pool
- Statement Cache
  - stores a partially compiled version of a statement in order to increase performance
  - Needs more memory
- Object or Result cache
  - First, second level
  - distributed

Caching 2



Additional (Hibernate) Tips

- See http://www.informit.com/articles/article.asp?p=353736
  - Use an SQL Monitor
  - Optimize Collections
  - Don't do bulk inserts
    - If use optimized hibernate parameters

Questions?

Thank you!!