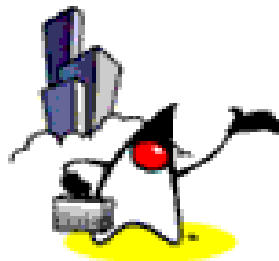


Hibernate Basics

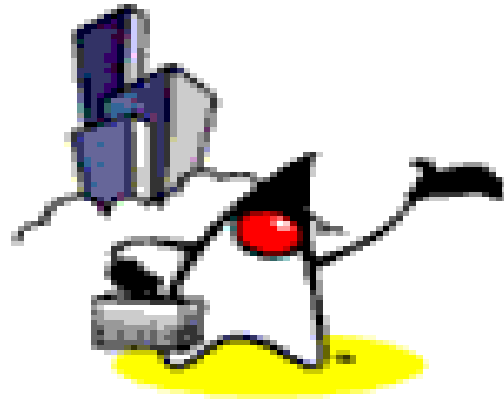


Topics

- Why use Object/Relational Mapping(ORM)?
- What is and Why Hibernate?
- Hibernate architecture
- Instance states
- Persistence lifecycle operations
- DAOs
- Transaction
- Configuration
- SessionFactory
- Persistent class

Topics covered in other presentations

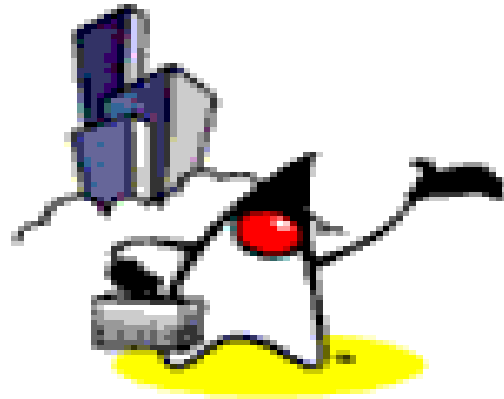
- Hibernate Step by Step
- Hibernate Criteria API
- Hibernate HQL
- Hibernate Mapping



Why Use ORM?

Why Object/Relational Mapping?

- A major part of any enterprise application development project is the persistence layer
 - Accessing and manipulate persistent data typically with relational database
- ORM handles Object-relational impedance mismatch
 - Data lives in the relational database, which is table driven (with rows and columns)
 - Relational database is designed for fast query operation of table-driven data
 - We want to work with objects, not rows and columns of table



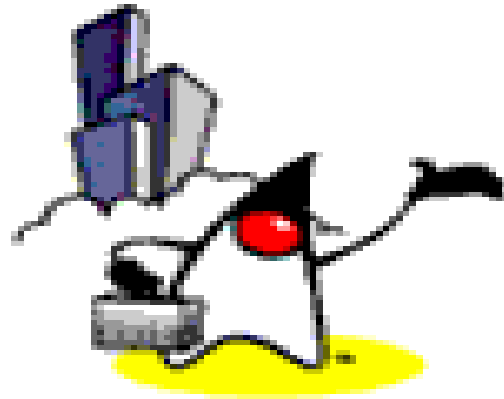
What is & Why Hibernate?

What is Hibernate?

- Object/relational mapping framework for enabling transparent POJO persistence
 - Let you work without being constrained by table-driven relational database model
- Build persistent objects following common OO programming concepts
 - Association
 - Inheritance
 - Polymorphism
 - Composition
 - Collection API for “many” relationship

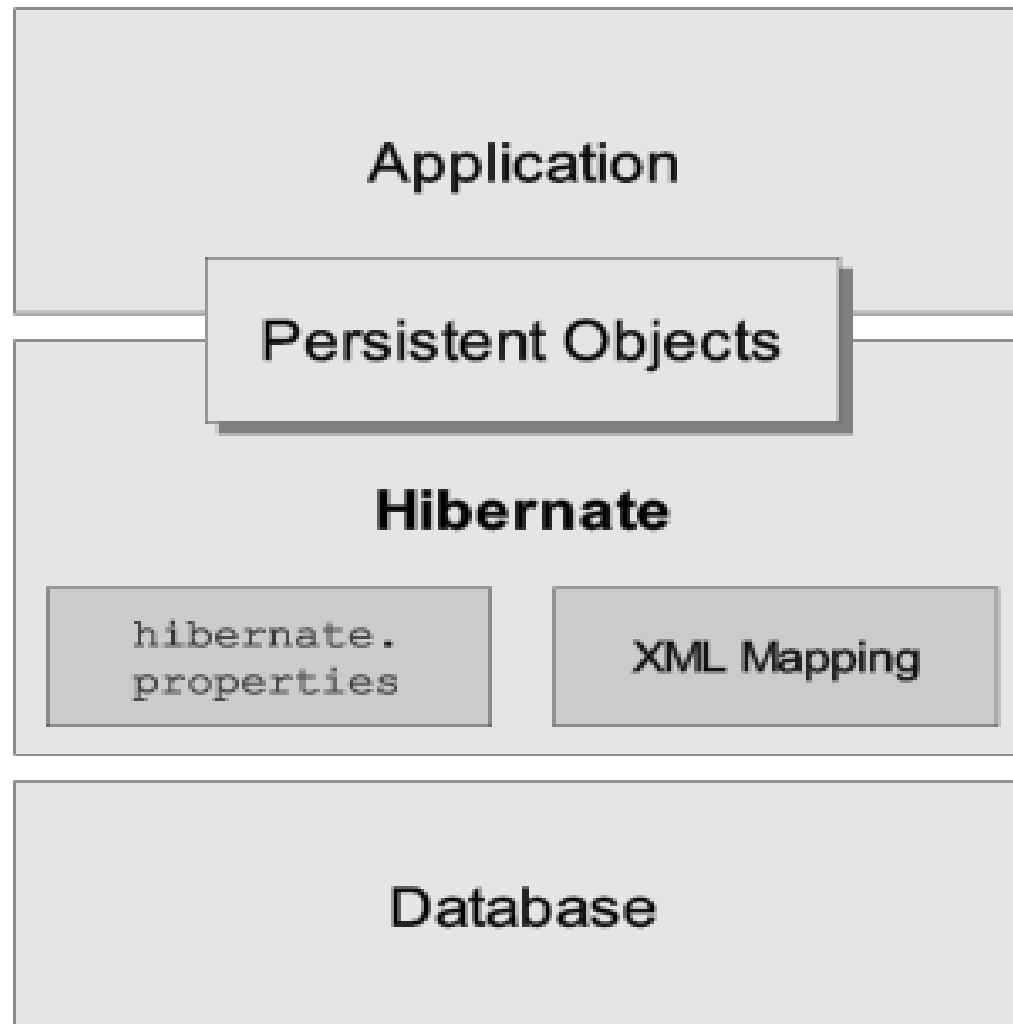
Why use Hibernate?

- Allows developers focus on domain object modelling not the persistence plumbing
- Performance
 - High performance object caching
 - Configurable materialization strategies
- Sophisticated query facilities
 - Criteria API and Query by Criteria
 - Query By Example (QBE)
 - Hibernate Query Language (HQL)
 - Native SQL



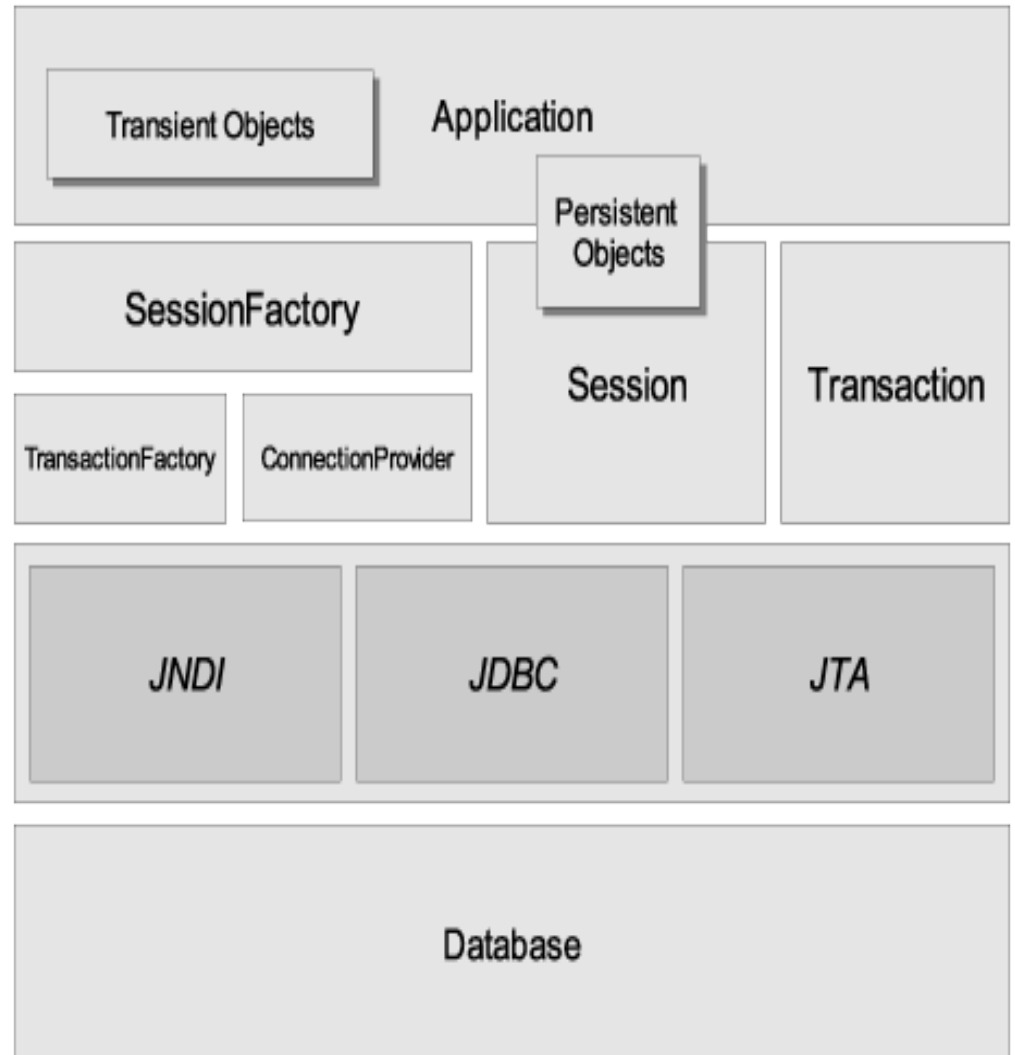
Hibernate Architecture

Hibernate Architecture



Hibernate Architecture

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



Hibernate Framework Objects

- SessionFactory
 - Represented by *org.hibernate.SessionFactory* class
 - A factory for *Session* and a client of *ConnectionProvider*
 - Typically one for each database
 - A threadsafe (immutable) cache of compiled mappings for a single database
 - Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level

Hibernate Framework Objects

- Session
 - Represented by *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

Hibernate Framework Objects

- Persistent objects and collections
 - Short-lived, single threaded objects containing persistent state and business function
 - These might be ordinary JavaBeans/POJOs, the only special thing about them is that they are currently associated with (exactly one) Session
 - Changes made to persistent objects are reflected to the database tables
 - As soon as the Session is closed, they will be detached and free to use in any application layer (e.g. directly as data transfer objects to and from presentation)

Hibernate Framework Objects

- Transient and detached objects
 - Instances of persistent classes that are not currently associated with a Session, thus without a persistent context
 - They may have been instantiated by the application and not (yet) persisted or they may have been instantiated by a closed Session
 - Changes made to Transient and detached objects do not get reflected to the database table

Hibernate Framework Objects

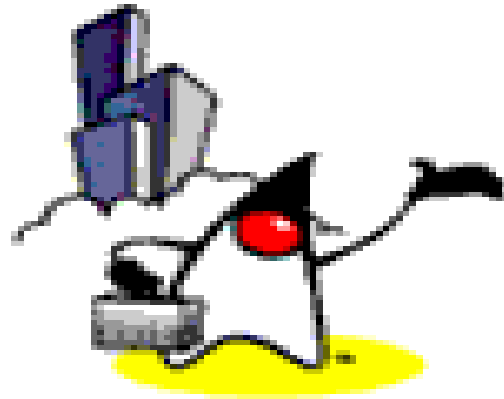
- Transaction
 - Represented by *org.hibernate.Transaction*
 - A single-threaded, short-lived object used by the application to specify atomic units of work
 - Abstracts application from underlying JDBC, JTA or CORBA transaction.
 - A Session might span several Transactions in some cases.
 - However,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

Hibernate Framework Classes

- ConnectionProvider
 - Represented by *org.hibernate.connection.ConnectionProvider*
 - A factory for (and pool of) JDBC connections.
 - Abstracts application from underlying *Datasource* or *DriverManager*.
 - Not exposed to application, but can be extended/implemented by the developer

Hibernate Framework Classes

- TransactionFactory
 - Represented by *org.hibernate.TransactionFactory*
 - A factory for Transaction instances.
 - Not exposed to the application, but can be extended/implemented by the developer



Instance States

Instance States

- An instance of a persistent classes may be in one of three different states, which are defined with respect to a persistence context
 - transient
 - persistent
 - detached
- The persistence context is represented by Hibernate Session object

Instance States

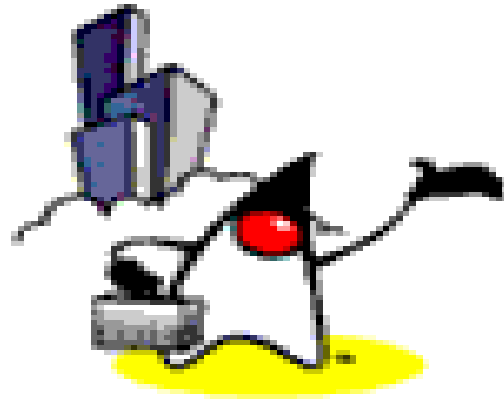
- “transient” state
 - The instance is not, and has never been associated with any session (persistence context)
 - It has no persistent identity (primary key value)
 - It has no corresponding row in the database
 - ex) When POJO instance is created
- “persistent” state
 - The instance is currently associated with a session (persistence context).
 - It has a persistent identity (primary key value) and a corresponding row in the database
 - ex) When POJO instance is persisted

Instance States

- “detached”
 - The instance was once associated with a persistence context, but that context was closed, or the instance was serialized to another process
 - It has a persistent identity and, perhaps, a corresponding row in the database
 - Used when POJO object instance needs to be sent over to another program for manipulation without having persistent context

State Transitions

- Transient instances may be made persistent by calling `save()`, `persist()` or `saveOrUpdate()`
- Persistent instances may be made transient by calling `delete()`
- Any instance returned by a `get()` or `load()` method is persistent
- Detached instances may be made persistent by calling `update()`, `saveOrUpdate()`, `lock()` or `replicate()`
- The state of a transient or detached instance may also be made persistent as a new persistent instance by calling `merge()`.



Persistence Lifecycle Operations

Lifecycle Operations

- Session interface provides methods for lifecycle operations
- Result of lifecycle methods affect the instance state
 - Saving objects
 - Loading objects
 - Getting objects
 - Refreshing objects
 - Updating objects
 - Deleting objects
 - Querying objects

Saving Objects

- Creating an instance of a class you map with Hibernate mapping does not automatically persist the object to the database until you save the object with a valid Hibernate session
 - An object remains to be in “transient” state until it is saved and moved into “persistent” state
- The class of the object that is being saved must have a mapping file (*myclass.hbm.xml*)
- Saving object generates *org.hibernate.event.SaveOrUpdateEvent*

Java methods for saving objects

- From Session interface
 - public Serializable save(Object object)
 - public void save(Object object, Serializable id)
 - public Serializable save(String entityName, Object object)

Example: Saving Objects

- Note that the Person is a POJO class with a mapping file (person.hbm.xml)

```
Person person = new Person();  
person.setName("Sang Shin");  
session.save(person);
```

Loading Objects

- Used for loading objects from the database
- Each *load(..)* method requires object's primary key as an identifier
 - The identifier must be *Serializable* – any primitive identifier must be converted to object
- Each *load(..)* method also requires which domain class or entity name to use to find the object with the id
- The returned object, which is returned as *Object* type, needs to be type-casted to the domain class

Java methods for loading objects

- From Session interface
 - `public Object load(Class theClass, Serializable id)`
 - `public Object load(String entityName, Serializable id)`
 - `public void load(Object object, Serializable id)`

Getting Objects

- Works like load() method

load() vs. get()

- Only use the load() method if you are sure that the object exists
 - load() method will throw an exception if the unique id is not found in the database
- If you are not sure that the object exists, then use one of the get() methods
 - get() method will return null if the unique id is not found in the database

Java methods for getting objects

- From Session interface
 - `public Object get(Class theClass, Serializable id)`
 - `public Object get(String entityName, Serializable id)`

Example: Getting Objects

```
Person person = (Person) session.get(Person.class, id);  
if (person == null){  
    System.out.println("Person is not found for id " + id);  
}
```

Refreshing Objects

- Used to refresh objects from their database representations in cases where there is a possibility of persistent object is not in sync. with the database representation
- Scenarios you might want to do this
 - Your Hibernate application is not the only application working with this data
 - Your application executes some SQL directly against the database
 - Your database uses triggers to populate properties on the object

Java methods for Refreshing objects

- From Session interface
 - `public void refresh(Object object)`

Updating Objects

- Hibernate automatically manages any changes made to the persistent objects
 - The objects should be in “persistent” state not transient state
- If a property changes on a persistent object, Hibernate session will perform the change in the database (possibly by queuing them first)
- From developer perspective, you do not have to any work to store these changes to the database
- You can force Hibernate to commit all of tis changes using flush() method
- You can also determine if the session is dirty through isDirty() method

Deleting Objects

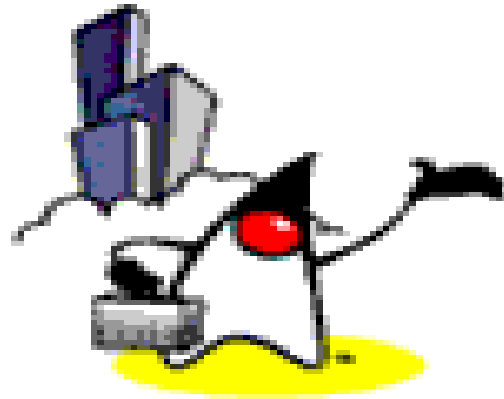
- Removes an object from the database

Querying Objects

- Hibernate provides 3 different ways of querying objects
 - Criteria API
 - HQL
 - Native SQL
- We will cover these query schemes in other presentation

Life-cycle Operations and SQL commands

- `save()` and `persist()` result in an SQL INSERT
- `delete()` results in an SQL DELETE
- `update()` or `merge()` result in an SQL UPDATE
- Changes to persistent instances are detected at flush time and also result in an SQL UPDATE
- `saveOrUpdate()` and `replicate()` result in either an INSERT or an UPDATE



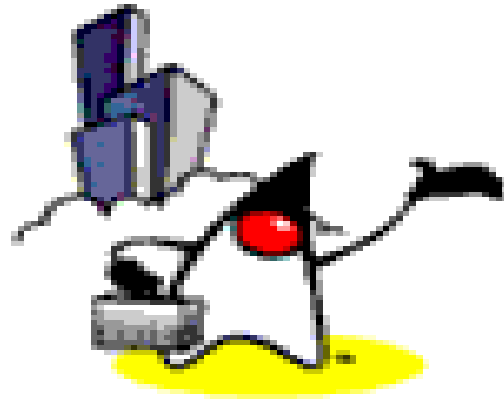
Cascading

Cascading Operations

- Lifecycle operations can be cascaded
- Cascading configuration flags (specified in the mapping file)
 - create
 - merge
 - delete
 - save-update
 - evict
 - replicate
 - lock
 - refresh

Cascading Operations Example

```
<hibernate-mapping >
  <class name="Event" table="events">
    <id name="id" column="uid" type="long">
      <generator class="increment"/>
    </id>
    <property name="name" type="string"/>
    <property name="startDate" column="start_date"
      type="date"/>
    <property name="duration" type="integer"/>
    <many-to-one name="location" column="location_id"
      class="Location" cascade="save-update" />
  </class>
</hibernate-mapping>
```



Persistent Classes

Persistent Classes

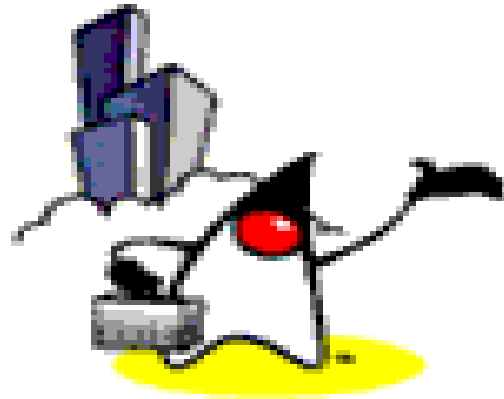
- Persistent classes are classes in an application that implement the entities of the business domain (e.g. Customer and Order in an E-commerce application)
- Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model.
- Hibernate3 assumes very little about the nature of your persistent objects
 - You may express a domain model in other ways: using trees of Map instances, for example.

How to write a POJO Class

- Rule 1: Implement a no-argument
 - All persistent classes must have a default constructor (which may be non-public) so that Hibernate can instantiate them using `Constructor.newInstance()`
- Rule 2: Provide an identifier property
 - This property maps to the primary key column of a database table.
 - The property might have been called anything, and its type might have been any primitive type, any primitive "wrapper" type, `java.lang.String` or `java.util.Date`
 - The identifier property is strictly optional. You can leave them off and let Hibernate keep track of object identifiers internally. We do not recommend this, however.

How to write a POJO Class

- Rule 3: Declare accessors and mutators for persistent fields
 - Hibernate persists JavaBeans style properties, and recognizes method names of the form getFoo, isFoo and setFoo



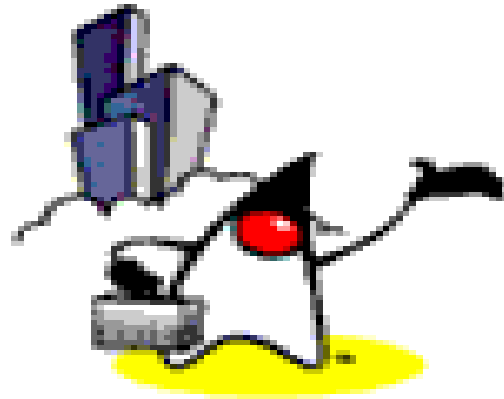
Data Access Objects (DAOs)

What is a DAO?

- DAO pattern
 - Separation of data access (persistence) logic from business logic
 - Enables easier replacement of database without affecting business logic
- DAO implementation strategies
 - Runtime pluggable through factory class (most flexible)
 - Domain DAO interface and implementation
 - Domain DAO concrete classes

Example: PersonDaoInterface

```
public class PersonDaoInterface {  
    public void create(Person p) throws SomeException;  
    public void delete(Person p) throws SomeException;  
    public Person find(Long id) throws SomeException;  
    public List findAll() throws SomeException;  
    public void update(Person p) throws SomeException;  
}
```

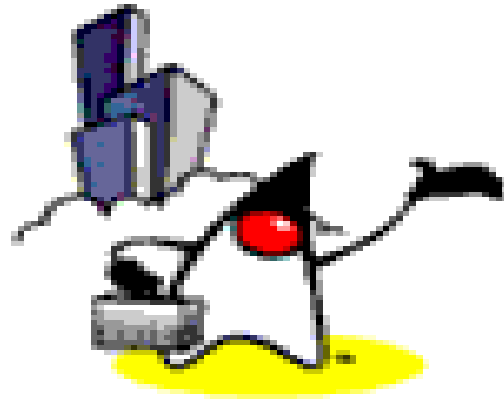


Transaction & Locking

Example: Transactional Operation

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Perform lifecycle operations

    tx.commit();
    tx = null;
} catch (HibernateException e){
    if (tx != null) tx.rollback();
} finally {
    session.close();
}
```



Hibernate Configuration

Two different ways of Configuring Hibernate

- Programmatic configuration
- XML configuration file
 - Specify a full configuration in a file named *hibernate.cfg.xml*
 - Can be used as a replacement for the *hibernate.properties* file or, if both are present, to override properties
 - By default, is expected to be in the root of your classpath
 - Externalization of the mapping file names to configuration is possible

XML Configuration File

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

  <!-- a SessionFactory instance listed as /jndi/name -->
  <session-factory
    name="java:hibernate/SessionFactory">

    <!-- properties -->
    <property
name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">>false</property>
    <property name="transaction.factory_class">
      org.hibernate.transaction.JTATransactionFactory
    </property>
    <property
name="jta.UserTransaction">java:comp/UserTransaction</property>
```

XML Configuration File

```
<!-- mapping files -->
```

```
<mapping resource="org/hibernate/auction/Item.hbm.xml"/>
```

```
<mapping resource="org/hibernate/auction/Bid.hbm.xml"/>
```

```
<!-- cache settings -->
```

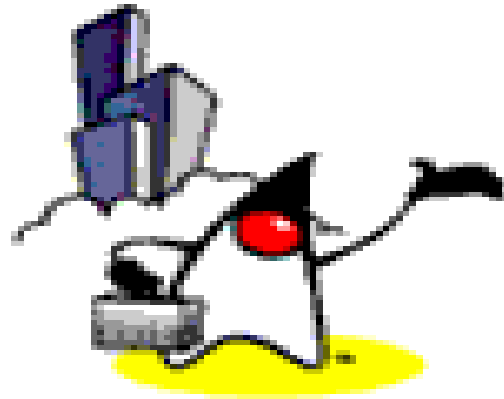
```
<class-cache class="org.hibernate.auction.Item" usage="read-write"/>
```

```
<class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
```

```
<collection-cache collection="org.hibernate.auction.Item.bids"  
usage="read-write"/>
```

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

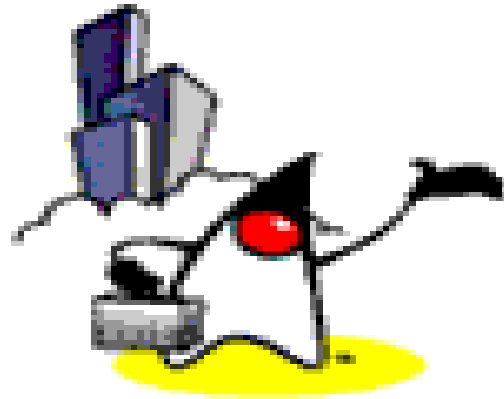
```
</hibernate-configuration>
```



Hibernate Mapping Files

Hibernate Mapping File

- Object/relational mappings are usually defined in an XML document.
- The mapping document is designed to be readable and hand-editable.
- The mapping language is Java-centric, meaning that mappings are constructed around persistent class declarations, not table declarations.
- Can be generated from the Java source using Annotation (from Hibernate 3.x only) or XDoclet
- Defines identifier generation, versioning, persistent properties, and object relationships and the mapping of these to the database



Mapping Classes to Tables

Mapping classes to tables

- The *class* element

```
<class name="domain.Answer"  
      table="answer"  
      dynamic-pdate="false"  
      dynamic-insert="false">
```

...

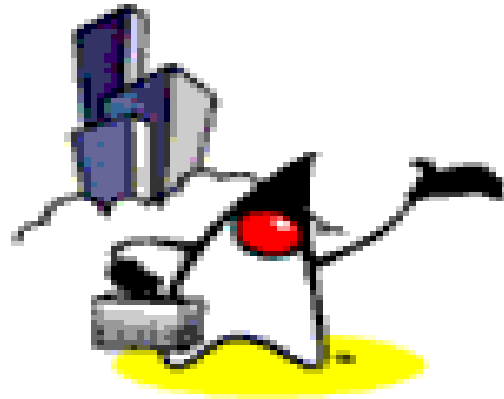
```
</class>
```

Attributes of <class>

<class

name="ClassName"	(1)
table="tableName"	(2)
discriminator-value="discriminator_value"	(3)
mutable="true false"	(4)
schema="owner"	(5)
catalog="catalog"	(6)
proxy="ProxyInterface"	(7)
dynamic-update="true false"	(8)
dynamic-insert="true false"	(9)
select-before-update="true false"	(10)
polymorphism="implicit explicit"	(11)
where="arbitrary sql where condition"	(12)
persister="PersisterClass"	(13)
batch-size="N"	(14)
optimistic-lock="none version dirty all"	(15)
lazy="true false"	(16)
entity-name="EntityName"	(17)
check="arbitrary sql check condition"	(18)
rowid="rowid"	(19)
subselect="SQL expression"	(20)
abstract="true false"	(21)
node="element-name"	

/>



Mapping Properties to Columns

Mapping object properties to columns

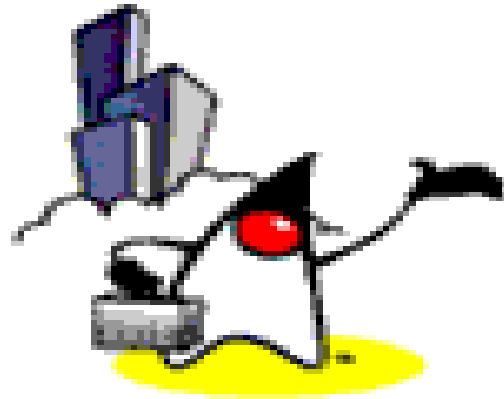
- Use *property* element

```
<property  
  name="reason"  
  type="java.lang.String"  
  update="true"  
  insert="true"  
  column="reason"  
  not-nul="true" />
```

Generating object identifiers

- Use the *id* element

```
<id name="id"  
    column="id"  
    type="java.lang.Long"  
    unsaved-value="null">  
    <generator class="native"/>  
</id>
```



Mapping Id Field

Mapping Id field

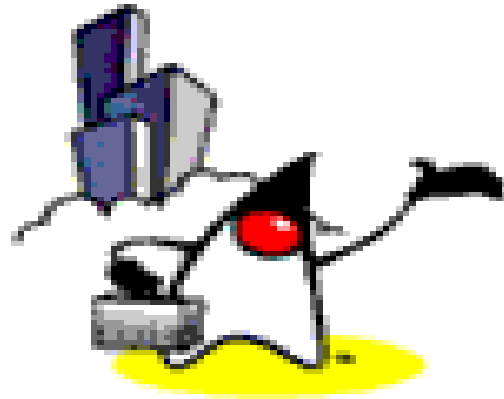
- Use *id* element
- Use generator subelement with class attribute, which specifies the key generation scheme

```
<class name="Person">
  <id name="id" type="int">
    <generator class="increment"/>
  </id>

  <property name="name" column="cname" type="string"/>
</class>
```

Key Generation Scheme via class attribute

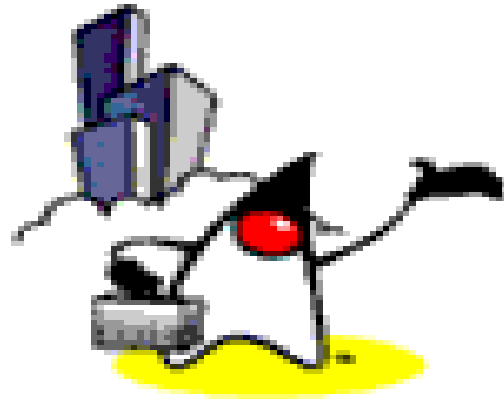
- class="increment"
 - It generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table. It should not be used in the clustered environment.
- class="assigned"
 - Lets the application to assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified.
- class="native"
 - It picks identity, sequence or hilo depending upon the capabilities of the underlying database.



Global Mapping

Attributes of <hibernate-mapping>

```
<hibernate-mapping
    schema="schemaName" (1)
    catalog="catalogName" (2)
    default-cascade="cascade_style" (3)
    default-access="field|property|ClassName" (4)
    default-lazy="true|false" (5)
    auto-import="true|false" (6)
    package="package.name" (7)
/>
```



Getting SessionFactory from XML configuration file

Getting SessionFactory using XML configuration file

- With the XML configuration, starting Hibernate is then as simple as

```
SessionFactory sf = new  
    Configuration().configure().buildSessionFactory();
```

- You can pick a different XML configuration file using

```
SessionFactory sf = new Configuration()  
    .configure("catdb.cfg.xml")  
    .buildSessionFactory();
```



Hibernate Basics

