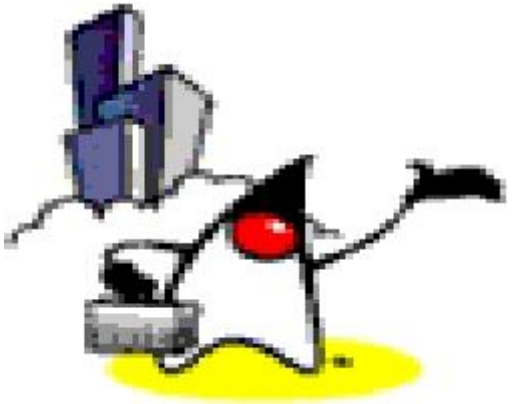
Hibernate Basics

Topics

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

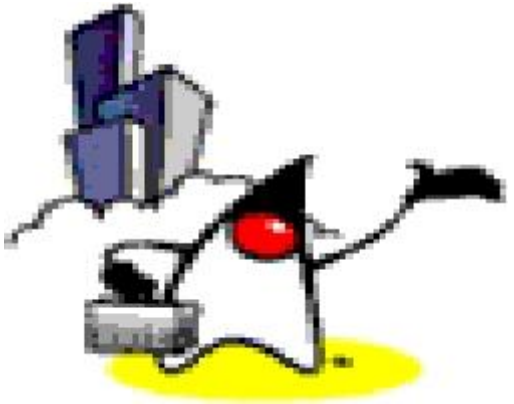
Why Use ORM?



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?



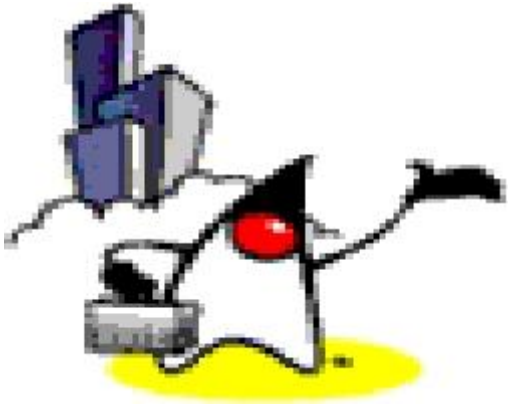
What is Hibernate?

- Object/relational mapping framework for enabling transparent POJO persistence
- Lets you build persistent objects following common OO programing concepts
  - Association
  - Inheritance
  - Polymorphism
  - Composition
  - Collection API for “many” relationship

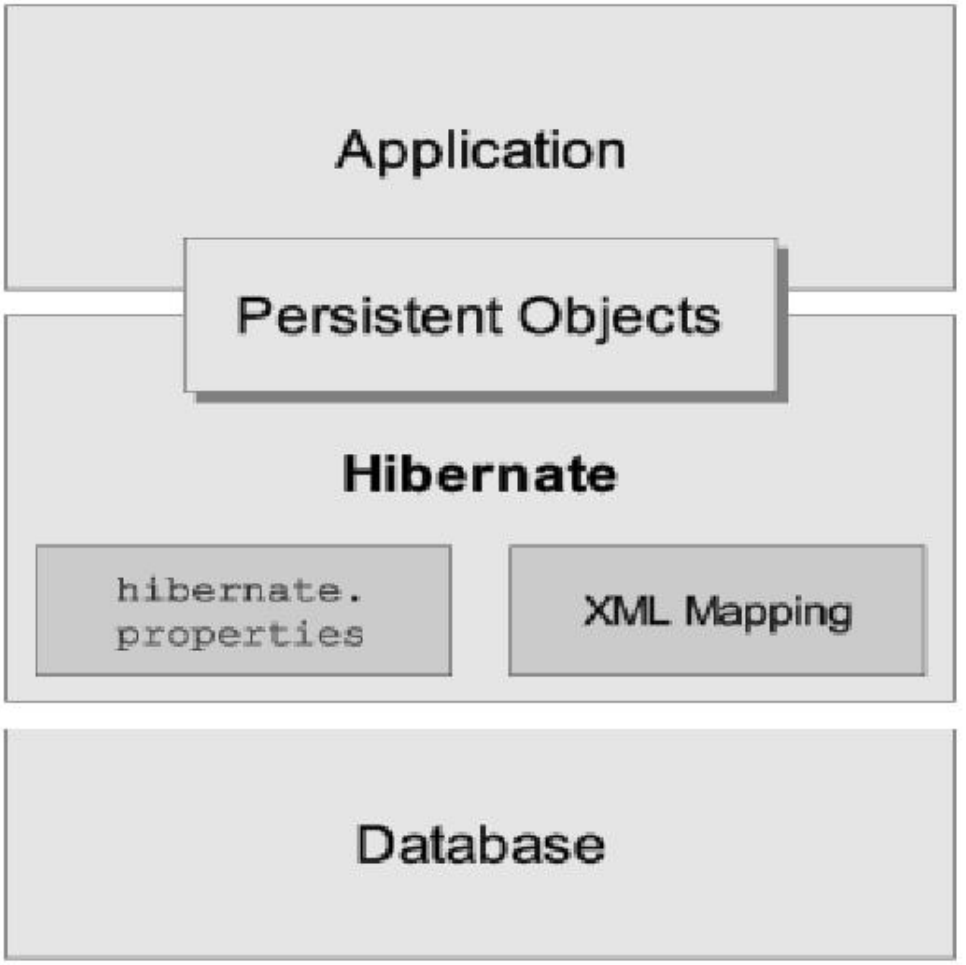
Why use Hibernate?

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

Hibernate Architecture

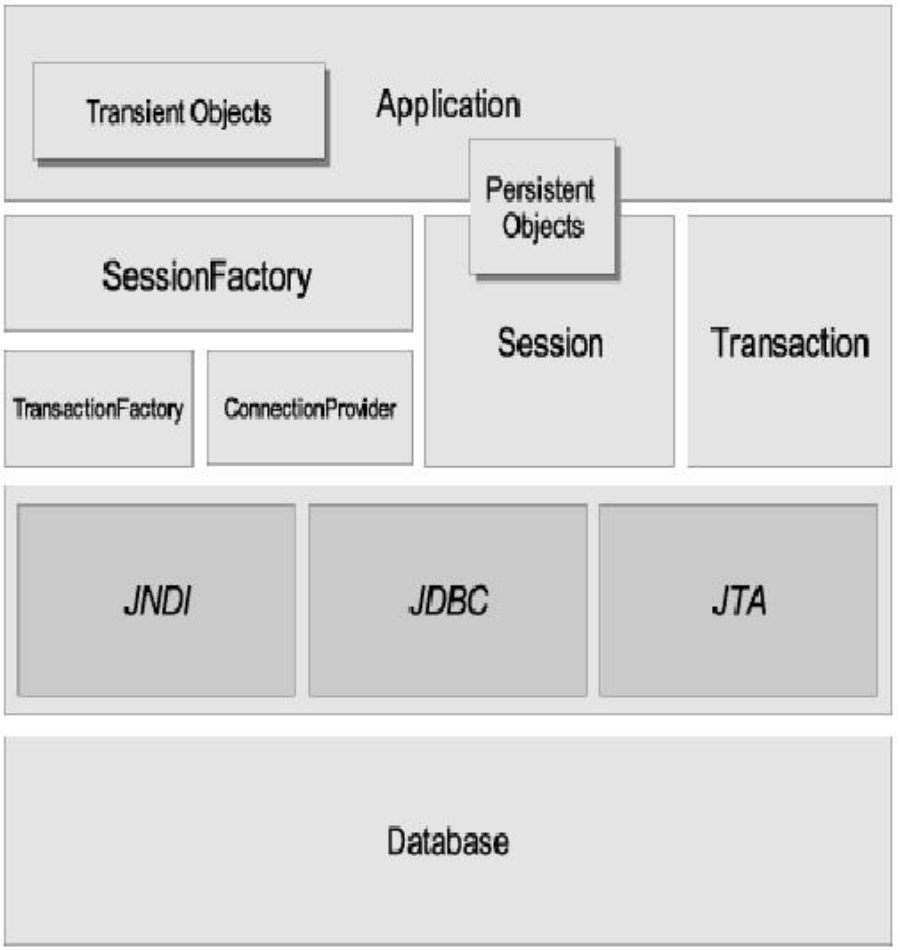


Hibernate Architecture



Hibernate Architecture

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

Hibernate Framework Objects

- Session
  - Represented by *org.hibernate.Session*
  - The life of a Session is bounded by the beginning and end of a logical transaction.
  - A session represents a persistence context
  - Handles life-cycle operations- create, read and delete operations - of persistent objects
  - A single-threaded, short-lived object representing a conversation between the application and the persistent store
  - Wraps a JDBC connection
  - Factory for *Transaction*

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 (when they are committed)
  - 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
    - They need to be persisted or merged before the change 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.

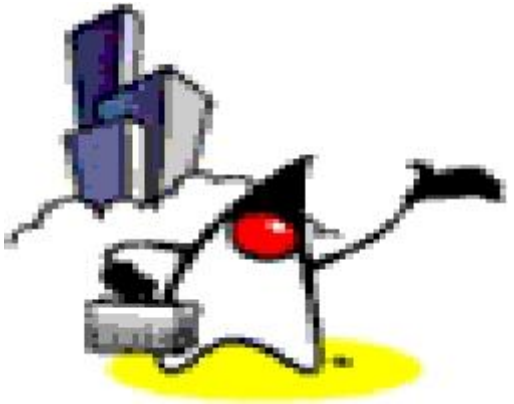
Hibernate Framework Classes

- 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 (does not belong to a persistence context)
  - persistent (belongs to a persistence context)
  - detached (used to belong to a persistence context)
- 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
  - When POJO instance is created outside of a session
- “persistent” state
  - The instance is currently associated with a session (persistence context).
  - It has a persistent identity (primary key value) and likely to have a corresponding row in the database
  - When an object is created within a session or a transient object gets persisted

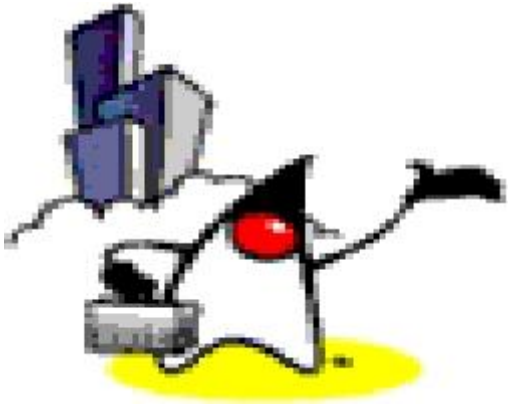
Instance States

- “detached”
  - The instance was once associated with a persistence context, but that context was closed
  - 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()*.

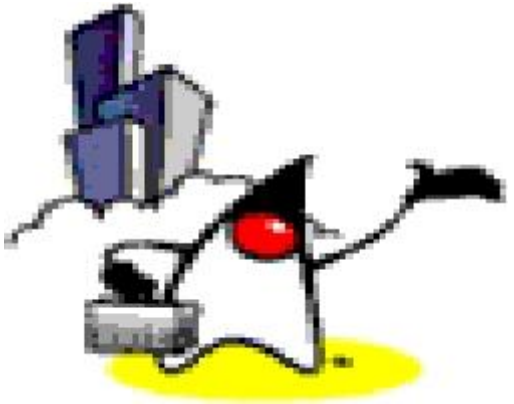
Methods of Session Interface



Types of Methods in Session Class

- Life cycle operations
- Transaction and Locking
- Managing resources
- JDBC Connection

Lifecycle Operations



Lifecycle Operations

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

Java methods for saving objects

- From Session interface

  // Persist the given transient instance,

  // first assigning a generated identifier.

  public Serializable save(Object object)

  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);

  // You can get an identifier

  Object identifier = session.getIdentifier(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 a domain class

Java methods for loading objects

- From Session interface

// Return the persistent instance of the given entity
// class with the given identifier, assuming that the
// instance exists.
public Object load(Class theClass, Serializable id)

// Return the persistent instance of the given entity
// class with the given identifier, obtaining the specified
// lock mode, assuming the instance exists.
public Object load(Class theClass, Serializable id, LockMode
lockMode)

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
  - *// Return the persistent instance of the given entity*
  - *// class with the given identifier, or null if there is no*
  - *// such persistent instance.*
  - *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
  - after executing direct SQL (eg. a mass update) in the same session
  - where a database trigger alters the object state upon insert or update

Java methods for Refreshing objects

- From Session interface

// Re-read the state of the given instance from the

// underlying database.

public void refresh(Object object)

public void refresh(Object object, LockMode lockMode)

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 when a transaction is committed (possibly by queuing the changes first)
- From developer perspective, you do not have to do any work to store these changes to the database
- You can force Hibernate to commit all of its changes using *flush()* method
- You can also determine if the session is dirty through *isDirty()* method. Return true if the session contains pending changes; false otherwise.

Deleting Objects

- From Session interface

// Remove a persistent instance from the datastore.
// The argument may be an instance associated with
// the calling Session or a transient instance with
// an identifier associated with existing persistent
// state. This operation cascades to associated
// instances if the association is mapped with
// cascade="delete".
public void delete(Object object)
public void delete(String entityName, Object object)

Replicating Objects

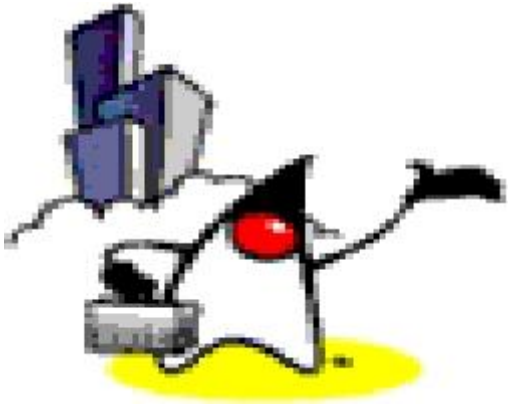
- From Session Interface

// Persist the state of the given detached instance,
// reusing the current identifier value.
public void replicate(Object object,
ReplicationMode replicationMode)

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*

POJO as Domain Classes



Domain Classes

- Domain 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.

Steps to write a Domain Class

- Step 1: Implement a no-argument
- Step 2: Provide an identifier property
- Step 3: Create accessor methods

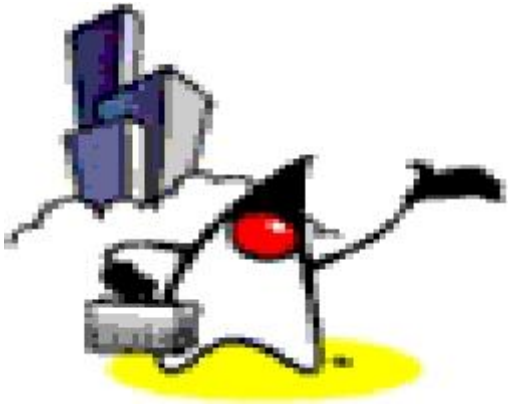
Steps to write a Domain Class

- Step 1: Implement a no-argument
  - All persistent classes must have a default constructor so that Hibernate can instantiate them using *Constructor.newInstance()*
- Step 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*
  - Composite key is possible
  - The identifier property is optional. You can leave them off and let Hibernate keep track of object identifiers internally – not recommended, however

Steps to write a POJO Class

- Step 3: Declare accessor methods for persistent fields
  - Hibernate persists JavaBeans style properties, and recognizes method names of the form getFoo, isFoo and setFoo

Composite Primary Key



Example: Composite Primary Key Mapping

```
<hibernate-mapping>
   <class name="org.example.composite.Application" table="APPLICATION">

      <!-- applicationPK is another class the implements Serializable -->

      <composite-id
       name="applicationPK"
       class="org.example.composite.ApplicationPK" >

       <key-property name="collectorateCode" column="COLLECTORATE_CODE" type="string" />
       <key-property name="applicationNumber" column="APPL_SNO" type="integer"/>
       <key-property name="year" column="YEAR_NBR" type="integer"/>

      </composite-id>

     <!-- Normal properties of a java class -->
      <property name="name" column="NAME" type="stringr"></property>
      <property name="age" column="AGE" type="integer"></property>

   </class>
</hibernate-mapping>
```

Example: Composite Key Class

```
public class ApplicationPK implements Serializable {

        // declare composite properties (actually
        // getter and setter methods)
        // declare constructor

}
```

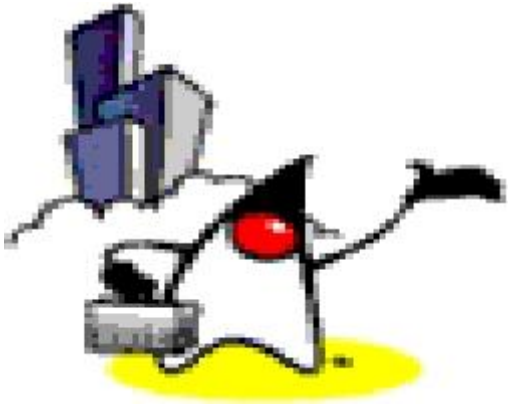
Example: Domain Class

```
public class Application {

        private ApplicationPK applicationPK;

        //declare other properties
        //declare constructor
}
```

Example: Lifecycle operation

```
// Create composite primary key
applicationPK = new ApplicationPK(...);

// Perform load operation
SessionObject.load(Application.class, applicationPK)
```

Data Access Objects (DAOs)



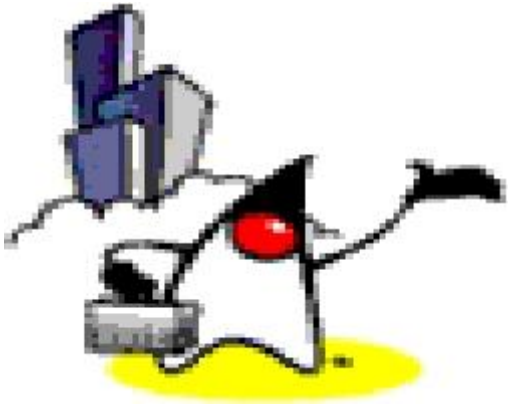
What is a DAO?

- DAO pattern
  - Separation of data access (persistence) logic from business logic
  - Enables easier replacement of database without affecting business logic
- DAO implementation strategies
  - Strategy 1: Runtime pluggable through factory class (most flexible)
  - Strategy 2: Domain DAO interface and implementation
  - Strategy 3: 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;
    public WhateverType
            whateverPersonRelatedMethod(Person p)
                                    throws SomeException;
}
```

Hibernate Configuration



Two different ways of Configuring Hibernate

- 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

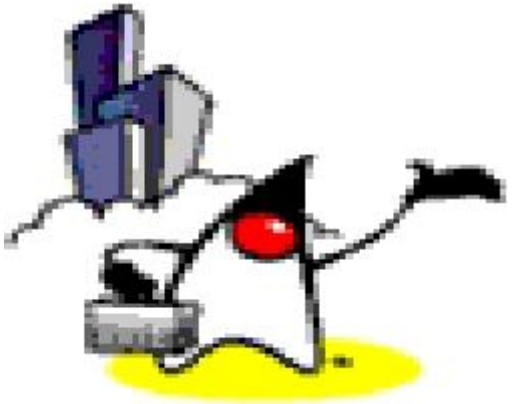
Hibernate Configuration File

```
<?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>
```

Hibernate 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>
```

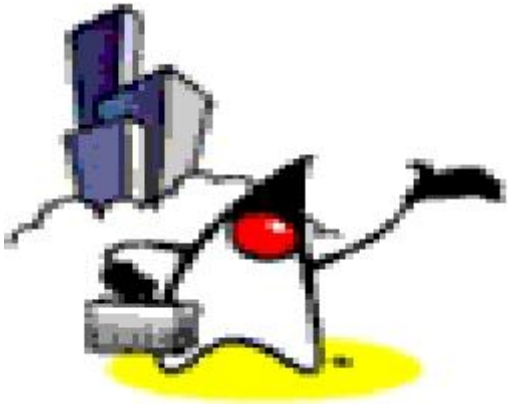
Hibernate Mapping Files



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-update="false"
        dynamic-insert="false">
...
</class>
```

Attributes of <class> element

<class
name="ClassName"
table="tableName"
discriminator-value="discriminator_value"
mutable="true|false"
schema="owner"
catalog="catalog"
proxy="ProxyInterface"
dynamic-update="true|false"
dynamic-insert="true|false"
select-before-update="true|false"
polymorphism="implicit|explicit"
where="arbitrary sql where condition"
persister="PersisterClass"
batch-size="N"
optimistic-lock="none|version|dirty|all"
lazy="true|false"
entity-name="EntityName"
check="arbitrary sql check condition"
rowid="rowid"
subselect="SQL expression"
abstract="true|false“
node="element-name"
/>

Attributes of <class> element

- *name*: The fully qualified Java class name of the persistent class (or interface).
- *table* (optional - defaults to the unqualified class name): The name of its database table.
- *discriminator-value* (optional - defaults to the class name): A value that distinguishes individual subclasses, used for polymorphic behaviour.

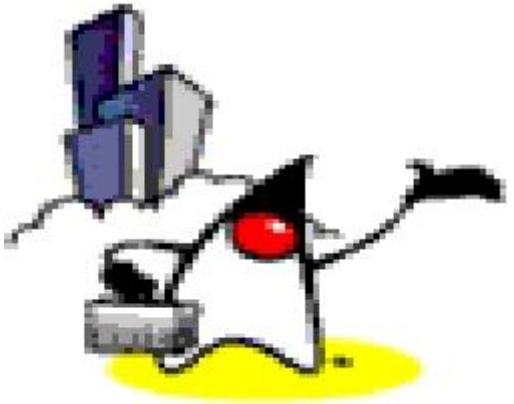
Attributes of <class>

- *dynamic-update* (optional, defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed.
- *dynamic-insert* (optional, defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.
- *optimistic-lock* (optional, defaults to version): Determines the optimistic locking strategy.

Attributes of <class>

- *lazy* (optional): Lazy fetching may be completely disabled by setting lazy="false".

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” />
```

Attributes of <property> element

```
<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"
    generated="never|insert|always"
    node="element-name|@attribute-name|element/@attribute|."
    index="index_name"
    unique_key="unique_key_id"
    length="L"
    precision="P"
    scale="S"
 />
```

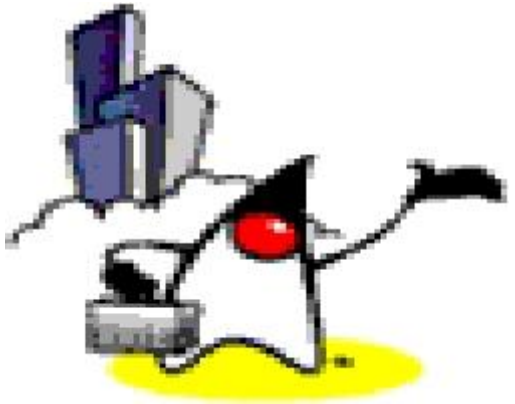
Attributes of <property> element

- *name: t*he name of the property, with an initial lowercase letter.
- *column* (optional - defaults to the property name): the name of the mapped database table column.
- *unique* (optional): Enable the DDL generation of a unique constraint for the columns.
- *not-null* (optional): Enable the DDL generation of a nullability constraint for the columns.

Attributes of <property> element

- *optimistic-lock* (optional - defaults to true): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.
- *generated* (optional - defaults to never): Specifies that this property value is actually generated by the database

Mapping Id Field



Mapping Id field

- Use *id* element
- Use *generator* sub-element 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>
```

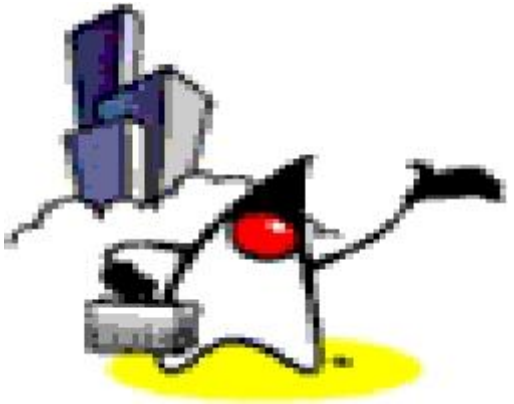
Key Generation Scheme via class attribute

- 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 the used in the clustered environment.
- class=”identity”
  - Supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL.
- class=”hilo”
  - Uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column

Key Generation Scheme via class attribute

- 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.
- class=”uuid”
  - Uses a 128-bit UUID algorithm to generate identifiers

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 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();
```

The End