Configuration conf = new Configuration();

conf.configure();

SessionFactorysf = conf.buidSessionFactory();

Session ss = sf.openSession();

Transaction tx = ss.beginTransaction();

Oye o = new Oye();

o.setName(“eswar”);

ss.save(o);

tx.commit();

ss.close();

When we call configure method hibernate reads info available in hbm and cfg files.

When we call buildSessionFactroy() is called hibernate decides about sql statements to deal with various tables and creates SessionFactory object.

When save() is called hibernate stores Oye object refered by the variable ‘a’ in its cache.

**Without Hibernate**

Databases are complex to implement. This is especially true when components of a database based on two different approaches (i.e. object oriented vs. relational) must interact with each other. Object Relational Mapping (ORM) frameworks can act as mediators between the two approaches. Hibernate, the topic of this article, is an ORM framework for Java.

The process of transformation from a relational approach to an object oriented approach is known as Object Relational Mapping

**Why do you need ORM tools like hibernate?**

The main advantage of ORM like hibernate is that it shields developers from messy SQL. Apart from this, ORM provides following benefits:

Improved productivity

High-level object-oriented API

Less Java code to write

No SQL to write

Improved performance

Sophisticated caching

Lazy loading

Eager loading

Improved maintainability

A lot less code to write

Improved portability

ORM framework generates database-specific SQL for you

**org.hibernate.Session**

A single-threaded, short-lived object representing aconversation between the application and thepersistent store

Wraps a JDBC connection

Factory for Transaction

Holds a mandatory (first-level) cache of persistentobjects, used when navigating the object graph orlooking up objects by identifier

**Instance states**

An instance of a persistent classes may be inone 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

**Transient state**

– The instance is not, and has never been associatedwith 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 acorresponding row in the database

– ex) When POJO instance is persisted

**Detached state**

– The instance was once associated with apersistence context, but that context was closed, orthe instance was serialized to another process

– It has a persistent identity and, perhaps, acorresponding row in the database

– Used when POJO object instance needs to be sentover to another program for manipulation withouthaving persistent context

Transient instances may be made persistent bycalling save(), persist() or saveOrUpdate()

Persistent instances may be made transient bycalling delete()

Any instance returned by a get() or load()method is persistent

Detached instances may be made persistent bycalling update(), saveOrUpdate(), lock() orreplicate()

The state of a transient or detached instancemay also be made persistent as a newpersistent instance by calling merge().

**The different types of Id generators supported by hibernate are:**

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.

identity: It supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.

sequence: The sequence generator uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int

hilo: The hilo generator uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate\_unique\_key and next\_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database. Do not use this generator with connections enlisted with JTA or with a user-supplied connection.

seqhilo: The seqhilo generator uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.

uuid: The uuid generator uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.

guid: It uses a database-generated GUID string on MS SQL Server and MySQL.

native: It picks identity, sequence or hilo depending upon the capabilities of the underlying database.

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.

select: retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.

foreign: uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.

**one-to-one associations:**

There are two varieties of one-to-one associations:

1. primary key associations
2. unique foreign key associations

Ensure that the primary keys of the related rows in the PERSON and EMPLOYEE tables are equal. You use a special Hibernate identifier generation strategy called foreign:

<class name="person" table="PERSON">

<id name="id" column="PERSON\_ID">

<generator class="foreign"><param name="property">employee</param></generator>

</id>

...

<one-to-one name="employee" class="Employee"constrained="true"cascade="save-update"/>/\*Here no column is specified. i.e no extra column is needed in Person table\*/

</class>

Alternatively, a foreign key with a unique constraint, from Employee to Person, can be expressed as:

*Employee.hbm.xml*

<many-to-one name="person" class="Person" column="PERSON\_ID" unique="true"/>/\* here one more column is needed in employee table \*/

This association can be made bidirectional by adding the following to the Person mapping:

*Person.hbm.xml*

<one-to-one name="employee" class="Employee" property-ref="person"/>

**many-to-many:**

Student.hbm.xml

<set name="courses" table="student\_courses">

<key column="student\_id"/>

<many-to-many column="course\_id" class="com.icesoft.icefaces.tutorial.crud.hibernate.Course"/>

</set>

Course.hbm.xml

<set name="students" table="student\_courses" inverse="true">

<key column="course\_id"/>

<many-to-many column="student\_id" class="com.icesoft.icefaces.tutorial.crud.hibernate.Student"/>

</set>

**many-to-one:**

1. uni-directional navigation

<many-to-one name="country" column="COUNTRY\_ID" class="org.demo.Country" not-null="true"/>

1. bi-directional navigation:

Person.hbm.xml

<many-to-one name="country"column="COUNTRY\_ID" class="org.demo.Country" not-null="true"/>

Country.hbm.xml

<set name="citizens" inverse="true" cascade="save-update">

<key column="COUNTRY\_ID"/>

<one-to-many class="org.demo.Person"/>

</set>

**Collection Mapping**

A <bag> is an unordered collection, which can contain duplicated elements. That means if you persist a bag with some order of elements, you cannot expect the same order retains when the collection is retrieved. There is not a “bag” concept in Java collections framework, so we just use aava.util.List to correspond to a <bag>.

A <set> is very similar to a <bag>. The only difference is that set can only store unique objects. That means no duplicated elements can be contained in a set. When you add the same element to a set for second time, it will replace the old one. A set is unordered by default but we can ask it to be sorted, which we will discuss later. The corresponding type of a <set> in Java is java.util.Set.

<set name="chapters" table="BOOK\_CHAPTER">

<key column="BOOK\_ID" />

<element column="CHAPTER" type="string" length="100" />

</set>

A <list> is an indexed collection where the index will also be persisted. That means we can retain the order of the list when it is retrieved. It differs from a <bag> for it persists the element index while a <bag> does not. The corresponding type of a <list> in Java is java.util.List. The table structure for a <list> has one more column, CHAPTER\_INDEX, than a <bag>. It is used for storing the element index. The type of this column is int.

An <array> has the same usage as a <list>. The only difference is that it corresponds to an array type in Java, not a java.util.List. It is seldom used unless we are mapping for legacy applications. In most cases, we should use <list> instead. That is because the size of an array cannot be increased or decreased dynamically, where a list can.

A <map> is very similar to a <list>. The difference is that a map uses arbitrary keys to index the collection, not an integer index using in a list. A map stores its entries in key/value pairs. You can lookup the value by its key. The key of a map can be of any data types. The corresponding type of a <map> in Java is java.util.Map.

|  |  |
| --- | --- |
| <map name="chapters" table="BOOK\_CHAPTER">  <key column="BOOK\_ID" />  <map-key column="CHAPTER\_KEY" type="string" />  <element column="CHAPTER" type="string" length="100" />  </map> | <set name="chapters" >  <key column="BOOK\_ID" />  <one-to-many class="Chapter" />  </set>  This is for to treat chapter as entity |

we can even turn off the lazy initialization for this collection

<bag name="chapters" table="BOOK\_CHAPTER" **lazy="false"**>

By default, the fetch mode of a collection is “select”. Hibernate will issue two SELECT statements for querying the book object and the chapter collection separately. We can change the fetch mode to “join” telling Hibernate to issue a single SELECT statement with table join.

<bag name="chapters" table="BOOK\_CHAPTER" lazy="false" **fetch="join"**>

Hibernate supports two ways of sorting a collection.

**Sorting in memory**

The first way of sorting a collection is utilizing the sorting features provided by the Java collections framework. The sorting occurs in the memory of JVM which running Hibernate, after the data being read from database. Notice that for large collections this kind of sorting may not be efficient. Only <set> and <map> supports this kind of sorting.

To ask a collection to be sorted in natural order, we define the sort attribute to be "natural" in the collection. Hibernate will use the compareTo() method defined in the java.lang.Comparableinterface to compare the elements. Many basic data types, such as String, Integer and Double haveimplemented this interface.

<set name="chapters" table="BOOK\_CHAPTER" sort="natural">

<set name="chapters" table="BOOK\_CHAPTER" **sort="UserdefinedComparator"**>//use your comparator

**Sorting in database**

If your collection is very large, it will be more efficient to sort it in the database. We can specify the

order-by condition for sorting this collection when retrieval. Notice that this order-by attribute

should be a SQL column, not a property name in Hibernate.

<set name="chapters" table="BOOK\_CHAPTER" order-by="CHAPTER">

**Table per class hierarchy**

<discriminator column="DISC\_TYPE" type="string" />

<subclass name="AudioDisc" discriminator-value="AUDIO">

<property name="numOfSongs" type="int" column="NUM\_OF\_SONGS" />

</subclass>

**Table per subclass**

<joined-subclass name="AudioDisc" table="AUDIO\_DISC">

<key column="DISC\_ID" />

<property name="numOfSongs" type="int" column="NUM\_OF\_SONGS" />

</joined-subclass>

**Table per concrete class**

<union-subclass name="AudioDisc" table="AUDIO\_DISC">

<property name="singer" type="string" column="SINGER" />

<property name="numOfSongs" type="int" column="NUM\_OF\_SONGS" />

</union-subclass>

How can a whole class be mapped as immutable?

Mark the class as mutable="false" (Default is true),. This specifies that instances of the class are (not) mutable. Immutable classes, may not be updated or deleted by the application.

dynamic-update (defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed

dynamic-insert (defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.

Retrieving object graphs efficiently is another central concern

The SQL INSERT statement contains the values that were held by the object at the point when save() was called. You can, of course, modify the object after calling save(), and your changes will be propagated to the database

as an SQL UPDATE.

**Updating the persistent state of a detached instance**

Modifying the user after the session is closed will have no effect on its persistent representation in the database. When the session is closed, user becomes a detached instance. It may be reassociated with a new Session by calling update() or lock().

It doesn’t matter if the object is modified before or after it’s passed to update(). The important thing is that the call to update() is used to reassociate the detached instance to the new Session (and current transaction) and tells Hibernate to treat the object as dirty

it does matter whether changes are made before or after the object is associated with the session. Changes made before the call to lock() aren’t propagated to the database; you only use lock() if you’re sure that the detached instance hasn’t been modified.

**Retrieving a persistent object**

If no row with the given identifier value exists in the database, the get() returns null.

**Making a detached object transient**

Finally, you can make a detached instance transient, deleting its persistent state from the database. This means you don’t have to reattach (with update() or lock()) a detached instance to delete it from the database; you can directly delete a detached instance:

Session session = sessions.openSession(); Transaction tx = session.beginTransaction();

session.delete(user); tx.commit(); session.close();

In this case, the call to delete() does two things: It associates the object with the Session and then schedules the object for deletion, executed on tx.commit().

**Transitive persistence** is a technique that allows you to propagate persistence to transient and detached subgraphs automatically. For example, if we add a newly instantiated Category to the already persistent hierarchy of categories, it should automatically become persistent without a call to Session.save().

Hibernate does not navigate an association when searching for transient or detached objects, so saving, deleting, or reattaching a Category won’t affect the child category objects. This is the opposite of the persistence-by-reachability

default behavior.

**cascade="save-update"** tells Hibernate to navigate the association when the transaction is committed and when an object is passed to save() or update() and save newly instantiated transient instances and persist changes to detached instances.

**cascade="all"** means to cascade both save-update and delete, as well ascalls to evict and lock.

**cascade="all-delete-orphan"** means the same as cascade="all" but, in addition, Hibernate deletes any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).

**Retrieving objects:**

|  |  |
| --- | --- |
| Load | Get |
| load() just returns a proxy by default and database won’t be hit until the proxy is first invoked.  load() method will throw an exception if the unique id is not found in the database.the exception would be thrown later, when the proxy is accessed.  Only use the load() method if you are sure that the object exists. | get() will hit the database immediately.  get() method will return null 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. |

**HQL** isn’t a data-manipulation language like SQL. It’s used only for object retrieval, not for updating, inserting, or deleting data. Object state synchronization is the job of the persistence manager, not the developer.

Query q = session.createQuery("from User u where u.firstname = :fname");

q.setString("fname", "Max"); List result = q.list();

**QBC:**Many developers prefer QBC, considering it a more object-oriented approach. They also like the fact that the query syntax may be parsed and validated at compile time, whereas HQL expressions aren’t parsed until runtime.

Criteria criteria = session.createCriteria(User.class); criteria.add( Expression.like("firstname", "Max") );

List result = criteria.list();

**Query by example:** A typical use case for QBE is a search screen that allows users to specify a range ofproperty values to be matched by the returned result set. This kind of functionalitycan be difficult to express cleanly in a query language; string manipulations wouldbe required to specify a dynamic set of constraints.

User exampleUser = new User(); exampleUser.setFirstname("Max");

Criteria criteria = session.createCriteria(User.class); criteria.add( Example.create(exampleUser) );

List result = criteria.list();

**Fetching strategies**:In traditional relational data access, you’d fetch all the data required for a particular computation with a single SQL query, taking advantage of inner and outer joins to retrieve related entities. Some primitive ORM implementations fetch data piecemeal, with many requests for small chunks of data in response to the application’s navigating a graph of persistent objects. This approach doesn’t make efficient use of the relational database’s join capabilities.

The problem of fetching object graphs efficiently (with minimal access to the database) has often been addressed by providing association-level fetching strategies specified in metadata of the association mapping.

**Immediate fetching**—The associated object is fetched immediately, using a sequential database read (or cache lookup).

**Lazy fetching**—The associated object or collection is fetched “lazily,” when it’s first accessed. This results in a new request to the database (unless the associated object is cached).

**Eager fetching**—The associated object or collection is fetched together with the owning object, using an SQL outer join, and no further database request is required.

**Batch fetching**—This approach may be used to improve the performance of lazy fetching by retrieving a batch of objects or collections when a lazy association is accessed. (Batch fetching may also be used to improve the performance of immediate fetching.)

**Caching theory**:

A cache keeps a representation of current database state close to the application, either in memory or on disk of the application server machine. The cache is a local copy of the data.

There are three main types of cache:

**Transaction scope**—Attached to the current unit of work, which may be an actual database transaction or an application transaction. It’s valid and used as long as the unit of work runs. Every unit of work has its own cache.

**Process scope**—Shared among many (possibly concurrent) units of work or transactions. This means that data in the process scope cache is accessed by concurrently running transactions, obviously with implications on transaction

isolation. A process scope cache might store the persistent instances themselves in the cache, or it might store just their persistent state in a disassembled format.The real downside to process-scoped identity is the need to synchronize access to persistent instances in the cache, resulting in a high likelihood of deadlocks.

**Cluster scope**—Shared among multiple processes on the same machine oramong multiple machines in a cluster. It requires some kind of remote processommunication to maintain consistency. Caching information has to be replicatedto all nodes in the cluster. For many (not all) applications, clusterscope caching is of dubious value, since reading and updating the cachemight be only marginally faster than going straight to the database.

**First-Level-Cache**:You don’t have to do anything special to enable the session cache. It’s always on and, for the reasons shown, can’t be turned off. Whenever you pass an object to save(), update(), or saveOrUpdate(), and whenever you retrieve an object using load(), find(), list(), iterate(), or filter(), that object is added to the session cache. When flush() is subsequently called, the tate of that object will be synchronized with the database. If you don’t want this synchronization to occur, or if you’re processing a huge number of objects and need to manage memory efficiently, you can use the evict() method of the Session to remove the object and its collections from the

first-level cache.To completely evict all objects from the session cache, call Session.clear().

Consider this frequently asked question: “I get an OutOfMemoryException when I try to load 100,000 objects and manipulate all of them. How can I do mass updates with Hibernate?” It’s our view that ORM isn’t suitable for mass update (or mass delete) operations. f you have a use case like this, a different strategy is almost always better: call a

stored procedure in the database or use direct SQL UPDATE and DELETE statements. Don’t transfer all the data to main memory for a simple operation if it can be performed more efficiently by the database. If your application is mostly mass operation use cases, ORM isn’t the right tool for the job!

**Seccond-level-cache**:The Hibernate second-level cache has process or cluster scope; all sessions share he same second-level cache. The second-level cache actually has the scope of a SessionFactory.

Hibernate uses this cache mainly to reduce the number of SQL queries it needs to generate within a given transaction. or example, if an object is modified several times within the same transaction, Hibernate will generate only one SQL UPDATE statement at the end of the transaction, containing all the modifications.

**CacheProviders**

**EHCache** is a fast, lightweight, and easy-to-use in-process cache. It supports read-only and read/write caching, and memory- and disk-based caching. However, it does not support clustering.

**OSCache** is another open-source caching solution. It is part of a larger package, which also provides caching functionalities for JSP pages or arbitrary objects. It is a powerful and flexible package, which, like EHCache, supports read-only and read/write caching, and memory- and disk-based caching. It also provides basic support for clustering via either JavaGroups or JMS.

**SwarmCache** is a simple cluster-based caching solution based on JavaGroups. It supports read-only or non-strict read/write caching (the next section explains this term). This type of cache is appropriate for applications that typically have many more read operations than write operations.

**JBossTreeCache** is a powerful replicated (synchronous or asynchronous) and transactional cache. Use this solution if you really need a true transaction-capable caching architecture.

**Caching Strategies**

Once you have chosen your cache implementation, you need to specify your access strategies. The following four caching strategies are available:

•**Read-only:** This strategy is useful for data that is read frequently but never updated. This is by far the simplest and best-performing cache strategy.

•**Read/write:** Read/write caches may be appropriate if your data needs to be updated. They carry more overhead than read-only caches. In non-JTA environments, each transaction should be completed when Session.close() or Session.disconnect() is called.

•**Non-strict read/write:** This strategy does not guarantee that two transactions won't simultaneously modify the same data. Therefore, it may be most appropriate for data that is read often but only occasionally modified.

**•Transactional:** This is a fully transactional cache that may be used only in a JTA environment.

Cache Read-only Nonstrict Read/write Read/write Transactional

EHCache Yes YesYes No

OSCache Yes YesYes No

SwarmCache Yes Yes No No

JBossTreeCacheYes No NoYes

<class name="Category" table="CATEGORY">

<cache usage="read-write"/>

**Hibernate Criteria**

HQL forces programmers to drop into an SQL-like syntax for certain operations. So it would seem that Hibernate developers still need to know SQL. Fortunately, Hibernate offers a set of APIs that map the SQL functionality to objects. This allows programmers to use an entirely object-oriented approach without falling back on a relational methodology.

There are five core APIs that are commonly used. 1.Criteria 2.Criterion3. Restrictions4. Projection5. Order

Criterion is the object-oriented representation of the "where" clause of a SQL query.

The Projection class is an object-oriented representation of query resultset projection in a Criteria query. In simpler terms, projection refers to the fields mentioned in the select clause of a query.

The Order class represents the "order by" clause of SQL. By using the asc() and desc() methods of this class, order can be imposed upon the Criteria resultselt.

SELECT O.\*, P.\* FROM ORDERS O, PRODUCT P WHERE O.ORDER\_ID=P.ORDER\_ID;

List orders = session.createCriteria(Order.class).setFetchMode(“products”,FetchMode.JOIN).list();

session.createCriteria(Order.class).add(Restrictions.eq(“orderId”,”1092”))

Projections.count(“id”);Projections.groupProperty(“id”)

**Javax.servlet.Servlet Interface**

public abstract void init(javax.servlet.ServletConfig arg0) throws javax.servlet.ServletException;

public abstract void service(javax.servlet.ServletRequest arg0, javax.servlet.ServletResponse arg1) throws javax.servlet.ServletException, java.io.IOException;

public abstract void destroy();

public abstract javax.servlet.ServletConfiggetServletConfig();

public abstract java.lang.StringgetServletInfo();

**GenericServlet abstract class:**service method is abstract method

public abstract class javax.servlet.GenericServlet implements javax.servlet.Servlet, javax.servlet.ServletConfig, java.io.Serializable

private**transient**javax.servlet.ServletConfigconfig;

publicjava.lang.StringgetInitParameter(java.lang.String name)

publicjava.util.EnumerationgetInitParameterNames();

publicjavax.servlet.ServletConfiggetServletConfig();//servlet interface method

publicjavax.servlet.ServletContextgetServletContext();

publicjava.lang.StringgetServletName()

**HttpServlet:**abstract class without abstract methods

public abstract class javax.servlet.http.HttpServlet extends javax.servlet.GenericServlet implements java.io.Serializable

protected long getLastModified(javax.servlet.http.HttpServletRequestreq);

protected void service(javax.servlet.http.HttpServletRequestreq, javax.servlet.http.HttpServletResponseresp) throws javax.servlet.ServletException, java.io.IOException;

public void service(javax.servlet.ServletRequestreq, javax.servlet.ServletResponse res) throws javax.servlet.ServletException, java.io.IOException;

**private** void maybeSetLastModified(javax.servlet.http.HttpServletResponseresp, long lastModified);

<web-app>

<context-param>

<param-name>gone</param-name>

<param-value>gvalue</param-value>

</context-param>

<session-config>

<session-timeout>2</session-timeout>//minits

</session-config>

<servlet>

<servlet-name>one</servlet-name>

<servlet-class>Sone</servlet-class>//<jsp-file>

<load-on-startup>1<load-on-startup>

<init-param>

<param-name>pone</param-name>

<param-value>pvalue</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>one</servlet-name>

<url-pattern>/tsone</url-pattern>

<servlet-mapping>

ServletConfigconfig = getServletConfig();

String sone=config.getInitParameter(“pone”);

ServletContext application = getServletContext();

String sone= application.getInitParameter(“gone”);

PrintWriter out = response.getWriter();

response.setContentType(“text/plain”)//”application/pdf”

response.setDateHeader(“Expires”,”ptime”);

String sone = request.getParametr(“txtName”);

String stwo[] = request.getParameterValues(“cboxCourse”);

session.getMaxInactiveInterval();

session.setMaxInactiveInterval(120);

session.invalidate();

session.getId();

session.isNew()

request.getSession(true);gives new if does not exist

request.getSession(false);gives null if does not exist

**JSP**

What are seven JSP elements

Templet text, scriptlet, declaration, jsp directives, jsp expressions, jsp action tags, jspcustome tags.

What are the implicit variables

Request, response, session, application, config, page(this), pageContext, out.

We can not use implicit variables as part of the methods provided in declaration. As the implicit variables are part of \_\_jspService().

We can provide (override) jspInit, jspDestroy as part of declaration.

<%@ page import=”java.util.\*” %>

<%@ include file=”one.jsp” %>static include, content is included at compiletime, only one servlet is exicuted

<jsp:include page=”/one.jsp” />dynamic include

**What are JSP Directives?**

jsp directives are used to avoid java code from jsp.

jsp directives are 1.page 2.include and 3.taglib.

**Attributes for page direcites are:** import,language,contentType,errorPage,isErrorPage,buffer,autoFlush,session,threadSafe,extends,info,pageEncoding and isELIngnored

Attributes for include directives arepage and file

Attributes for Taglib directive areuri,prefix and tagdir

<%@ page autoFlush="false"%>( default is true)

Instead of sending the data byte by byte to the browser the web container collects the data in a buffer. If the size of the buffer is enough to accommodate the whole o/p of JSP page the o/p is collected and send at once to the browser. This reduces the amount of Network traffic. If the jsp generates more amount of data than the size of the buffer the

container frees the buffer once it is filled (this is done if autoFlush is set to true)

<jsp:useBean id=”xx” scope=”page” class=”org.Cone”/>

<jsp:setProperty name=”xx” property=”name”/>// property=”\*”

<jsp:getProperty name=”xx” property=”name”/>

how can we retrieve the values in jsp by using custom tags?

Tag handler class is not a servlet or jsp, so it doesn'thave access to many implicit objects which are available injsp. It, however, has a reference to PageContext using which

we can call :

pageContext.getAttribute(name) // looks in page scope only

pageContext.getAttribute(name,scope) // looks in specifiedscope

alternatively we can also also get references to otherscopes using :

pageContext.getRequest()

pageContext.getSession()

pageContext.getServletContext()

pageContext.getServletConfig()

And yes, this pageContext implicit object is given to thetag class by container, during initialisation of this tagclass when setPageContext(PageContext) is called by container

Javax.servlet.jsp.tagext.Tag

Javax.servlet.jsp.tagext.TagSupport

Opus.tld

<taglib>

<tag>

<name>hello</name>

<tag-class>org.Hello</tag-class>

<attribute>

<name>lend</name><required>true</required>

</attribute>

<tag>

<uri>[http://sun.com/lib </uri](http://sun.com/lib%20%3c/uri)>

<tlib-version>1.0</tlib-version>

<jsp-version>1.1</jsp-version>

</taglib>

Web.xml

<tlib><taglib-uri><http://sum.com/lib></taglib-uri>

<taglib-location>/web-inf/opus.tld</taglib-location></tlib>

Public intdoStartTag() throws JspException

Public intdoEndTag() throws JspException

EVAL\_PAGE: the remaining part of the page will be evaluated.

SKIP\_PAGE: the remaining part of the page will be skipped

SKIP\_BODY: the body of the tag will be skipped.

EVAL\_BODY\_INCLUDE: the body will be evaluated and output will be sent to the client as part of the response.

EVAL\_BODY\_BUFFERED: the body will be evaluated and output will be buffered.

Driver drv = new oracle.jdbc.driver.OracleDriver();

DriverManager.registerDriver(drv);

Connection con = DriverManager.getConnection();

Statement st = con.createStatement();

St.exicuteUpdate(sql);

Preparedstatementps = con.prepareStatement(sql);

Ps.setInt(1,5);

Ps.exicuteQuery();

CallableStatement call = con.prepareCall(“ { call pname(?,?)}”);

Call..setInt();

Call.registerOutParameter(1,Tpes.INTEGER);

Call.exicute();

**executeQuery():**The method, executeQuery(), takes a SQL String as an argument and returns a ResultSet object. This method should be used for any SQL calls that expect to return data from the database.

**executeUpdate( ) :**Update statements, on the other hand, are executed using the executeUpdate( ) method. This method returns the number of affected rows.

**execute()**  the Statement class provides an execute() method for situations in which you do not know whether the SQL being executed is a query or update. This usually happens when the application is executing dynamically created SQL statements. If the statement returns a row from the database, the method returns true. Otherwise it returns false. The application an then use the getResultSet() method to get the returned row.

**java.sql.ResultSet:** A ResultSetis one or more rows of data returned by a database query. The class simply provides a series of methods for retrieving columns from the results of a database query. The methods for getting a column all take the form:

*type*get *type*(int | String) in which the argument represents either the column number or column name desired. A nice side effect of this design is that you can store values in the database as one type and retrieve them as a completely different type. For example, if you need a Date from the database as a String, you can get it as a String by calling result\_set.getString(1) instead of esult\_set.getDate(1). Because the ResultSetclass handles only a single row from the database at any given time.JDBC 1.x allowed only one-way navigation through rows from a query. For JDBC 2.0, Sun added support for scrollable result sets.

conn.createStatement(ResultSet.TYPE\_FORWARD\_ONLY, ResultSet.CONCUR\_READ\_ONLY);

JDBC defines three types of result sets: TYPE\_FORWARD\_ONLY , TYPE\_SCROLL\_SENSITIVE , and

TYPE\_SCROLL\_INSENSITIVE .

**SQL NULL Versus Java null:**SQL and Java have a serious mismatch in handling null values. Specifically, using methods like getInt(), a Java ResultSet has no way of representing a SQL NULL value for any numeric SQL column. After retrieving a value from a ResultSet, it is therefore necessary to ask the ResultSet if the retrieved value represents a SQL NULL. For Java object types, a SQL NULL will often map to Java null. To avoid running into database oddities, however, it is recommended that you always check for SQL NULL. Checking for SQL NULL involves a single call to the wasNull( ) method in your ResultSet after you retrieve a value. The wasNull() method will return true if the last value read by a call to a getXXX() method was a SQL NULL. If, for example, your database allowed NULL values for PET\_COUNT column because you do not know the number of pets of all your customers, a call to getInt() could return some driver attempt at representing NULL, most likely 0. So how do you know in Java who has pets and who has an unknown number of pets? A call to wasNull() will tell you if represents an actual in the database or a NULL value in the database.

**Clean Up:**In the examples provided so far, you may have noticed many objects being closed through a close() method. The Connection, Statement, and ResultSet classes all have close( ). A given JDBCimplementation may or may not require you to close these objects before reuse. But some might require it, since they are likely to hold precious database resources. It is therefore always a good idea to close any instance of these objects when you are done with them. It is useful to remember that closing a Connection implicitly closes all Statement instances associated with the Connection.[3] Similarly, closing a Statement implicitly closes ResultSet instances associated with it. If you do manage to close a Connection before committing with auto-commit off, any uncommitted transactions will be lost.

**setAutoCommit( )** :Each Connection is separate, and a commit on one has no effect on the statements on another. The Connection class provides the setAutoCommit( ) method so you can turn auto-commit off.

**Batch Processing:** Using the JDBC 2.0 batch facilities, you can assign a series of SQL statements to a JDBC Statement (or one of its subclasses) to be submitted together for execution by the database.

Statement stmt = conn.createStatement( );

int[] rows;

for(int i=0; i<accts.length; i++) {

accts[i].calculateInterest( );

stmt.addBatch("UPDATE account " +"SET balance = " +accts[i].getBalance( ) +"WHERE acct\_id = " + accts[i].getID( ));

}

rows = stmt.executeBatch( );

This method returns an array of row counts of modified rows. The first element, for example, contains the number of rows affected by the first statement in the batch. This example uses the default auto-commit state in which each update is committed automatically.[1] If an error occurs somewhere in the batch, all accounts before the error will have their new balance stored in the database, and the subsequent accounts will not have had their interest calculated.(So set setAutoCommit(false)) The main difference is that a batch prepared or callable statement represents a single SQL statement with a list of parameter groups, and the database should create a query plan only once.

public abstract interface javax.ejb.EJBObject extends java.rmi.Remote

void remove() throws java.rmi.RemoteException, javax.ejb.RemoveException

Object getPrimaryKey() throws java.rmi.RemoteException;

EJBHomegetEJBHome() throws java.rmi.RemoteException

Handle getHandle() throws java.rmi.RemoteException;

booleanisIdentical(javax.ejb.EJBObject arg0) throws java.rmi.RemoteException;

public abstract interface javax.ejb.EJBHome extends java.rmi.Remote

void remove(java.lang.Object arg0) throws java.rmi.RemoteException, javax.ejb.RemoveException;

void remove(javax.ejb.Handle arg0) throws java.rmi.RemoteException, javax.ejb.RemoveException;

HomeHandle getHomeHandle() throws java.rmi.RemoteException;

EJBMetaData getEJBMetaData() throws java.rmi.RemoteException;

public abstract interface javax.ejb.MessageDrivenBean extends javax.ejb.EnterpriseBean

voidsetMessageDrivenContext(javax.ejb.MessageDrivenContext arg0) throws javax.ejb.EJBException;

voidejbRemove() throws javax.ejb.EJBException;

public abstract interface javax.jms.MessageListener

public abstract void onMessage(javax.jms.Message arg0);

public abstract interface javax.ejb.SessionBean extends javax.ejb.EnterpriseBean

public abstract void ejbActivate() throws javax.ejb.EJBException, java.rmi.RemoteException;

public abstract void ejbPassivate() throws javax.ejb.EJBException, java.rmi.RemoteException;

public abstract void ejbRemove() throws javax.ejb.EJBException, java.rmi.RemoteException;

public abstract void setSessionContext(javax.ejb.SessionContext arg0) throws javax.ejb.EJBException, java.rmi.RemoteException;

public abstract interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean

voidejbActivate() throws javax.ejb.EJBException, java.rmi.RemoteException;

voidejbPassivate() throws javax.ejb.EJBException, java.rmi.RemoteException;

voidejbLoad() throws javax.ejb.EJBException, java.rmi.RemoteException;

voidejbStore() throws javax.ejb.EJBException, java.rmi.RemoteException;

voidsetEntityContext(javax.ejb.EntityContext arg0) throws javax.ejb.EJBException, java.rmi.RemoteException;

voidunsetEntityContext() throws javax.ejb.EJBException, java.rmi.RemoteException;

**ENTITY BEAN:**

When a new bean is created, a new record must be inserted into the database and a bean instance must be associated with that data. As the bean is used and its state changes, these changes must be synchronized with the data in the database: entries must be inserted, updated, and removed. The process of coordinating the data represented by a bean instance with the database is called persistence.

Fields that are mapped to the database are called container-managed fields. Container-managed fields can be any Java primitive type or serializable objects. Most beans will use Java primitive types when persisting to a relational database, since it's easier to map Java primitives to relational data types.

**CMP**

Container-managed entity beans are the simplest to develop because they allow you to focus on the business logic, delegating the responsibility of persistence to the EJB container.

The advantage of container-managed persistence is that the bean can be defined independently of the database used to store its state. Container-managed beans can take advantage of a relational database or an object-oriented database. The bean state is defined independently, which makes the bean more reusable and flexible across applications.

The disadvantage of container-managed beans is that they require sophisticated mapping tools to define how the bean's fields map to the database.

**Message driven bean:**

A message-driven bean enables asynchronous clients to access the business logic in the EJB tier.

InitialContextctx = new InitialContext();

Queue queue = (Queue) ctx.lookup("jms/Queue");

QueueConnectionFactory factory = (QueueConnectionFactory) ctx.lookup("ConnectionFactory");

QueueConnection connection = factory.createQueueConnection();

QueueSession session = connection.createQueueSession(false, QueueSession.AUTO\_ACKNOWLEDGE);

QueueSender sender = session.createSender(queue);

**How can I call one EJB from inside of another EJB?**

EJBs can be clients of other EJBs. It just works. Use JNDI to locate the Home Interface of the other bean, then acquire an instance reference, and so forth.

**Why do we have a remove method in both EJBHome and EJBObject?**

With the EJBHome version of the remove, you are able to delete an entity bean without first instantiating it (you can provide a PrimaryKey object as a parameter to the remove method). The home version only works for entity beans. On the other hand, the Remote interface version works on an entity bean that you have already instantiated. In addition, the remote version also works on session beans (stateless and statefull) to inform the container of your loss of interest in this bean.

SELECT DISTINCT column\_name(s) FROM table\_name;

SELECT \* FROM Persons WHERE FirstName=Tove; we need to give ‘Tove’

SELECT \* FROM Persons WHERE Year='1965' wrong we need to give 1965 only

SELECT column\_name(s) FROM table\_name ORDER BY column\_name(s) ASC|DESC

SELECT \* FROM Persons WHERE FirstName='Tove' AND LastName='Svendson' OR

INSERT INTO table\_name VALUES (value1, value2, value3,...)

INSERT INTO table\_name (column1, column2, column3,...) VALUES (value1, value2, value3,...)

UPDATE Persons SET Address='Nissestien 67', City='Sandnes' WHERE LastName='Tjessem' AND FirstName='Jakob'

DELETE FROM table\_name WHERE some\_column=some\_value

DELETE FROM table\_nameorDELETE \* FROM table\_name

SELECT \*FROM PersonsWHERE ROWNUM <=5

SELECT \* FROM PersonsWHERE City LIKE '%tav%'

SELECT \* FROM PersonsWHERE LastName LIKE '[!bsp]%' starts with none of the three.

SELECT \* FROM PersonsWHERE FirstName LIKE '\_la' start with any char followed by ‘la’

SELECT \* FROM PersonsWHERE LastName IN ('Hansen','Pettersen')

SELECT \* FROM PersonsWHERE LastNameNOT BETWEEN 'Hansen' AND 'Pettersen'

**P\_Id LastName FirstName Address City O\_IdOrderNo P\_Id**

1 Hansen Ola Timoteivn 10 Sandnes1 77895 3

2 Svendson Tove Borgvn 23 Sandnes3 22456 1

3 Pettersen Kari Storgt 20 Stavanger

SELECT Persons.LastName, Orders.OrderNoFROM PersonsINNER JOIN OrdersON Persons.P\_Id=Orders.P\_Id

ORDER BY Persons.LastName

SELECT Persons.LastName, Orders.OrderNoFROM PersonsLEFT JOIN OrdersON Persons.P\_Id=Orders.P\_Id

SELECT Persons.LastName, Orders.OrderNoFROM PersonsFULL JOIN OrdersON Persons.P\_Id=Orders.P\_Id

SELECT Persons.LastName,Orders.OrderNoINTO Persons\_Order\_BackupFROM PersonsINNER JOIN Orders

ON Persons.P\_Id=Orders.P\_Id

CREATE TABLE Persons(P\_Idint NOT NULL,LastNamevarchar(255) NOT NULL)

CREATE TABLE Persons(P\_Idint NOT NULL UNIQUE,FirstNamevarchar(255));

CREATE TABLE Persons(P\_Idint NOT NULL,LastNamevarchar(255) NOT NULL,

CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName));

ALTER TABLE PersonsADD UNIQUE (P\_Id);

ALTER TABLE PersonsDROP CONSTRAINT uc\_PersonID;

CREATE TABLE Persons( \_Id int NOT NULL PRIMARY KEY);

ALTER TABLE PersonsDROP CONSTRAINT pk\_PersonID

CREATE TABLE Orders(O\_Idint NOT NULL PRIMARY KEY,P\_Idint FOREIGN KEY REFERENCES Persons(P\_Id))

ALTER TABLE OrdersADD FOREIGN KEY (P\_Id)REFERENCES Persons(P\_Id)

ALTER TABLE OrdersDROP CONSTRAINT fk\_PerOrders

CREATE TABLE Persons(P\_Idint NOT NULL CHECK (P\_Id>0))

ALTER TABLE PersonsADD CHECK (P\_Id>0)

Constraints are used to limit the type of data that can go into a table.

What is difference between Path and Classpath?

Path and Classpath are operating system level environment variales. Path is used define where the system can find the executables(.exe) files and classpath is used to specify the location .class files.

Why is the main method declared static?

main method is called by the JVM even before the instantiation of the class hence it is declared as static.

Can a main method be overloaded?

Yes. You can have any number of main methods with different method signature and implementation in the class.

**cloning** is the copying of an object to create an identical object. The method clone() in Object makes a copy of the object by copying each field reference and primitive to the new object. In essence this is a shallow object. Implementors can override this to make a deeper copy of the object if necessary. Objects like String do not need to be copied since they are immunitable.

Deepcloing is done in two ways 1 serilization and 2 cloning all related objects.

**Composition:** Life time of objects got to be the same. For example there is a composition between car-frame and a car. If the car gets destroyed the car-frame would also get destroyed.

**Aggregation**: Life times of the objects are not the same, one object can live even after the other object has been destroyed.

**Method Override:**if below condition fails exception occurs at compile time

Subclass should contain same return type or subclass to the super class return type.

Subclass should contain same exception or subclass to super class exception or none.

Access specifier same or more.

If super class and base class has same **static** method what will be the output for below code

Base b= new Super()

b.mone()//base class method will be called.

Base b = new Child();

s.o.p(b.x)//prints base variable value not chindi.ewith out creating object to base we are able to call base class instances

s.o.p(b.mone()) // child method is called

s.o.p(b.mone());// base class method will be called if it is static method

Java.lang.Throwable

**Compilation Exception:**CalssNotFoundException, CloneNotSupportedException, IterruptedException, NoSuchFeidException, IllegalAccessException

**Runtime Exception:**ArithmeticException, ClassCastException, NullPointerException, BufferOverflowException

**Catch syntax:**

Catch(subclassException se){}

Catch(SuperClassException se){}

**Collections:**

Collectios.sort(List); Collections.binarysearch(list,sting) ; reverse(List); swap(); rotate()

Arrays.asList(array); Object[] obj = list.toArray();

The advantage of hashing is that it allows the execution time of basic operations, such as add(), contains(), remove(), and size(), to remain constant for large sets.

Xxx =Intiger.parseXXX(“s”); string to primitive

Xxx = obj.xxxValue(); Wraper to primitive

ArrayList and vector default size is ten.

ArrayList and Vector, both are growable objects.

We can define load factor in Vector only.

Vector is synchronized one where ArrayList is not synchronized one.

ArrayList<Hell> al = new ArrayList(); // no compilation error

Hell hel1 = new Hell();

l.add(hel1);

al.add("String"); // it is compilation error

for(Hell h : al) { System.out.println(h); }

Set s = new HashSet();

s.add(hel1);

**for**(Object o : s) {

Hell h = (Hell)o;

System.*out*.println(h); }

HashSetadd() calls hashCode() and equals() methods(), but equals() is not called while adding two references points same object and also when haseCode is different equals() is not called

Object class toString() method internally calls hashCode();

Auto boxing is casting primitives into objects implicitly.

public interface Comparable<T>

publicint compareTo(T o);

Collection.sort() converts list as Array then calls Array.sort(), Array sort method creates duplicate array with clone then sorts using swap, then Collection sorts list with the help of listIterater.

ListIterator li = y.listIterator();

for (int j=0; j<y.size(); j++) {

li.next();

li.set("a");//it replace the list element values

}

Set add() returns treue or false. When we add duplicate elements it will give false.

Map put() returns null or value for duplicate keyes.

**Thread:**It is not abstract class but we need to override run(), It will not give compilation error if we don’t override also

Thread is independent path of execution.

Threads allow multiple activities to proceed concurrently in the same program.

Threads share the same address space.

Context switching b/w threads is less expensive than between processes.

The synchronized keyword ensures that only a single thread will execute a statement or lock at a time.

Synchronization is required for reliable communication between threads as well as for mutual exclusion.

The volatile modifier guarantees that any thread that reads a field willssee the most recently written value.

Public interface Runnable { public abstract void run(); }

**Public class Thread implements Runnable**

public static native void sleep(long millis) throws InterruptedException; MAX\_PRIORITY = 10

public static native void yield(); MIN\_PRIORITY = 1

public final native booleanisAlive(); NORM\_PRIORITY = 5

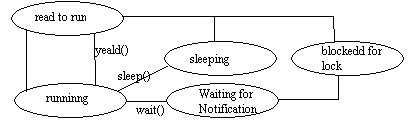
public final void setDaemon(boolean on) {}

public static native Thread currentThread();

public final booleanisDaemon() {} getName(), setName(), getState()

**Sleep:** It is Thread class method. We can call this method from anywhere. It will not release locks which are hold by this

**Wait:** It is object class method. This method can be executed on the object whose lock the thread holds, otherwise, the call will result in an IllegalMoniterStateException. It release lock on that object



Some examples of marker interfaces in the Java API include:

- java,lang.Cloneable

- java,io.Serializable

- java.util.EventListener

Marker interfaces are also called "tag" interfaces since they tag all the derived classes into a category based on their purpose. For example, all classes that implement the Cloneable interface can be cloned (i.e., the clone() method can be called on them). The Java compiler checks to make sure that if the clone() method is called on a class and the class implements the Cloneable interface. For example, consider the following call to the clone() method on an object o:

SomeObject o = new SomeObject();

SomeObject ref = (SomeObject)(o.clone());

If the class SomeObject does not implement the interface Cloneable (and Cloneable is not implemented by any of the superclasses that SomeObject inherits from), the compiler will mark this line as an error. This is because the clone() method may only be called by objects of type "Cloneable." Hence, even though Cloneable is an empty interface, it serves an important purpose.

One common problem occurs while using marker interfaces is that when a class implements them, all of its subclasses inherit them as well. Since you cannot unimplement an interface in Java, therefore a subclass that does not want to treat differently will still be marked as Marker.