Index

# [Difference between FetchType LAZY and EAGER in Java persistence?](#diff_fetchType_lazy_eager)

# [What is proxy in the context of load() method of Hibernate?](#what_is_proxy)

1. [Strategy ID generation](#Identity_Generation)
2. [Diff session.getLoad() and session.get()](#get_load)
3. [Cache Levels](#CACHEL_LEVELS)
4. [Maintain history](#history_tables)
5. [Hibernate mapping](#Hibernate_mapping)
6. [Hibernate relationships](#hibernatemapping)
7. [Diff between load and get() methods on session](#get_load)
8. [Diff between persist and save](#persist_and_save)

Hibernate 4

https://www.javacodegeeks.com/2015/03/hibernate-tutorial.html

in .cfg.xml

<properties>

<h2.version>1.3.176</h2.version>

<hibernate.version>4.3.8.Final</hibernate.version>

</properties>

  public void run() {

        SessionFactory sessionFactory = null;

        Session session = null;

        try {

            Configuration configuration = new Configuration();

            configuration.configure("hibernate.cfg.xml");

            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();

            sessionFactory = configuration.buildSessionFactory(serviceRegistry);

            session = sessionFactory.openSession();

            persistPerson(session);

        } catch (Exception e) {

            LOGGER.log(Level.SEVERE, e.getMessage(), e);

        } finally {

            if (session != null) {

                session.close();

            }

            if (sessionFactory != null) {

                sessionFactory.close();

            }

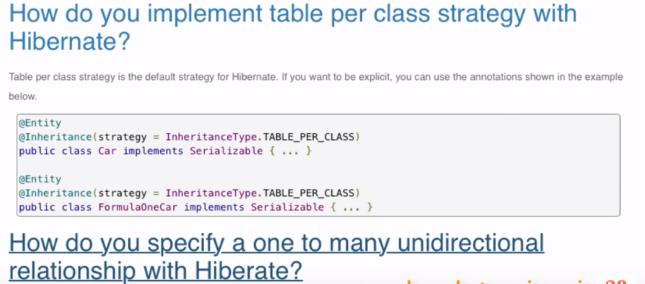
        }

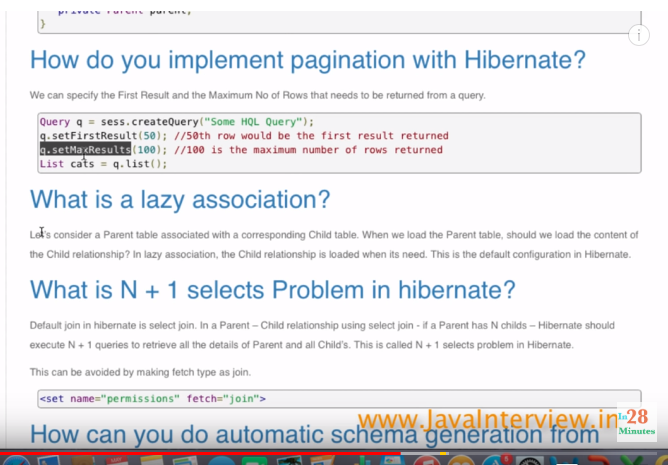
    }

    ..

After the table has been created, the invocation of session.save() issues an insert statement to the database. As Hibernate internally uses a PreparedStatement, we do not see the values on the console. In case you also want to see the values that are bound to the parameters of the PreparedStatement, you can set the logging level for the logger org.hibernate.type to FINEST. This is done within a file called logging.properties with the following content (the path to the file can be given for example as a system property -Djava.util.logging.config.file=src/main/resources/logging.properties):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | .handlers = java.util.logging.ConsoleHandler  .level = INFO    java.util.logging.ConsoleHandler.level = ALL  java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter    org.hibernate.SQL.level = FINEST  org.hibernate.type.level = FINEST |





[Identity\_Generation](#Identity_Generation)

Spring ORM module

The Spring Framework provides integration with Hibernate, JDO, Oracle TopLink, iBATIS SQL Maps and JPA: in terms of resource management, DAO implementation support, and transaction strategies.

Use Hibernate tools plugin to generate the hibernate config and mapping files

<http://www.journaldev.com/2696/spring-interview-questions-and-answers#spring-overview>

Difference between JPA and hibernate:

<http://stackoverflow.com/questions/9881611/whats-the-difference-between-jpa-and-hibernate>

JPA is just a specification, meaning there is no implementation. You can annotate your classes as much as you would like with JPA annotations, however without an implementation nothing will happen. Think of JPA as the guidelines that must be followed or an interface, while Hibernate's JPA implementation is code that meets the API as defined by the JPA specification and provides the under the hood functionality.

When you use Hibernate with JPA you are actually using the Hibernate JPA implementation. The benefit of this is that you can swap out Hibernate's implementation of JPA for another implementation of the JPA specification. When you use straight Hibernate you are locking into the implementation because other ORMs may use different methods/configurations and annotations, therefore you cannot just switch over to another ORM.

# [Relation between JPA and Hibernate annotation](http://stackoverflow.com/questions/15040583/relation-between-jpa-and-hibernate-annotation)

<http://stackoverflow.com/questions/15040583/relation-between-jpa-and-hibernate-annotation?rq=1>

As others have pointed out, JPA is an specification an Hibernate provides an implementation. You use JPA annotations/API and, by including Hibernate jars in your classpath, Hibernate will provide the actual logic.

Additionally, Hibernate offers an API that is unrelated to JPA. You can use that as well; the main difference is:

* if you use the JPA API, you may later replace Hibernate by other JPA implementation (v.g. EclipseLink) and you will need no changes to your program
* if you use directly Hibernate API, you have no implementation alternatives. The advantage may be that you can use features that Hibernate has defined but that are not part of JPA standard, which might be useful to you.

At any rate, what you should completely avoid is mixing JPA with the Hibernate API. As a beginner, I would advice you to stick to JPA.

To ensure that you are using JPA, only include classes/annotations from java.persistence. Do not include anything from org.hibernate (or, if you want just to use Hibernate, do just the opposite).

1. How to get hibernate connection in hibernate and how to interact with database?

The Spring **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.

Spring provides integration with Hibernate 3 and 4 under the form of HibernateTemplate

The [Spring Reference suggests this usage](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/orm.html#orm-hibernate-straight):

public class ProductDaoImpl implements ProductDao {

@Autowired

private SessionFactory sessionFactory;

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

public Collection loadProductsByCategory(String category) {

return this.sessionFactory.getCurrentSession()

.createQuery(

"from test.Product product where product.category=?")

.setParameter(0, category)

.list();

}

}

That way your classes don't have any dependencies to Spring, you just use plain Hibernate.

In Daoimpl java

@Repository

@Transactional

**public** **class** RMDAOImpl **implements** iRMDAO {

**private** Log logger = LogFactory.*getLog*(**this**.getClass());

**protected** org.springframework.orm.hibernate3.HibernateTemplate

template = **null**;

@Resource(name = "ctcSessionFactory")

**protected** SessionFactory sessionFactory;

@Autowired

**public** **void** init(SessionFactory sessionFactory) {

setSessionFactory(sessionFactory);

}

**public** **void** setSessionFactory(SessionFactory sessionFactory) {

template = **new** org.springframework.orm.hibernate3.HibernateTemplate

HibernateTemplate(sessionFactory);

}

}

In DAO method we can use the template like below.

List<FinCfgRmSector> instance = (List<FinCfgRmSector>) template.find("from FinCfgRmSector sector where sector.sectorId = ?", sectorId);

**if** (instance.size()>0) {

sector = instance.get(0);

} **else** {

logger.debug("Lookup of id ["+sectorId+"] found no sector");

**throw** **new** SectorNotFoundError("Sector with Id[" + sectorId + "] not found!");

}

Java bean like below.

\*\*

\* FinCfgRmFactor generated by hbm2java

\*/

@SuppressWarnings({ "serial"})

@XStreamAlias("FinCfgRmFactor")

@JsonAutoDetect

@Entity

@Table(name = "FIN\_CFG\_RM\_FACTOR")

**public** **class** FinCfgRmFactor **implements** java.io.Serializable {

Other methods:

template.save(version);

template.update(version);

**\*HibernateTemplate is** Helper class that simplifies Hibernate data access code. Automatically converts HibernateExceptions into DataAccessExceptions, following the org.springframework.dao exception hierarchy.

In pom.xml

[**-**](file:///C:\workspaceJboss1\ctc_2013_05_22\ctc\ctc-core-services\pom.xml) <dependency>

  <groupId>org.hibernate</groupId>

  <artifactId>hibernate-core</artifactId>

  <version>3.6.3.Final</version>

  </dependency>

1. what is session factory in hibernate?

SessionFactory is an interface and extends Referenceable, Serializable

*Session* is also interface

Creates Sessions. Usually an application has a single SessionFactory. Threads servicing client requests obtain Sessions from the factory.

Implementors must be threadsafe.

SessionFactorys are immutable. The behaviour of a SessionFactory is controlled by properties supplied at configuration time. These properties are defined on Environment.

1. How to get the hibernate sessionfactory from spring





Contex.xml under resource folder

<bean id=*"ctcSessionFactory"* class=*"org.springframework.orm.hibernate3.annotation.****AnnotationSessionFactoryBean****"*>

<property name=*"dataSource"* ref=*"ctcDataSource"* />

<property name=*"packagesToScan"* value=*"com.moodys.mis.ctc.models,com.moodys.mis.cfg.rm.models"* />

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.connection.driver\_class"*>${hibernate.connection.driver.class}</prop>

<prop key=*"hibernate.dialect"*>${hibernate.dialect}</prop>

<prop key=*"hibernate.show\_sql"*>${hibernate.show\_sql}</prop>

<prop key=*"hibernate.current\_session\_context\_class"*>${hibernate.current\_session\_context\_class}</prop>

<prop key=*"hibernate.cache.use\_second\_level\_cache"*>false</prop>

<prop key=*"hibernate.query.factory\_class"*>org.hibernate.hql.classic.ClassicQueryTranslatorFactory

</prop>

<prop key=*"hibernate.validator.apply\_to\_ddl"*>false</prop>

<prop key=*"hibernate.validator.autoregister\_listeners"*>false</prop>

</props>

</property>

</bean>

Use of dialect

Databases implement subtle differences in the SQL they use. Things such as data types for example vary across databases (e.g. in Oracle I might put an integer value in a number field and in SQL Server use an int field). Or database specific functionality - selecting the top n rows is different depending on the database. The dialect abstracts this so you don't have to worry about it.

Sybase dialect:

org.hibernate.dialect.SybaseDialect

#### . Oracle 9i/10g

org.hibernate.dialect.Oracle9Dialect

#### Microsoft SQL Server

org.hibernate.dialect.SQLServerDialect

Transaction management using in the same –context.xml

<bean id=*"ctcTransactionManager"*

class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*

p:sessionFactory-ref=*"ctcSessionFactory"* />

<tx:annotation-driven transaction-manager=*"ctcTransactionManager"* />

</beans>

In DAO impl class we can access using

@Resource(name = "ctcSessionFactory")

**protected** SessionFactory sessionFactory;

here **@Resource** The Resource annotation marks a resource that is needed by the application. This annotation may be applied to an application component class, or to fields or methods of the component class. When the annotation is applied to a field or method, the container will inject an instance of the requested resource into the application component when the component is initialized. If the annotation is applied to the component class, the annotation declares a resource that the application will look up at runtime.

@Resource vs @Autowired

difference is:

* @Resource allows you to specify a name of the injected bean
* @Autowired allows you to mark it as non-mandatory.
* @Resource means get me a **known resource by name**. The name is extracted from the name of the annotated setter or field, or it is taken from the name-Parameter.
* @Inject or @Autowired try to wire in a **suitable other component by type**

In daoimpl, get hibernate template by using the following.

@Autowired

**public** **void** init(SessionFactory sessionFactory) {

setSessionFactory(sessionFactory);

}

**public** **void** setSessionFactory(SessionFactory sessionFactory) {

template = **new** HibernateTemplate(sessionFactory);

}

1. Right way to use Hibernate sessionfactory in spring

<http://stackoverflow.com/questions/5104765/hibernatedaosupport-is-not-recommended-why>

Using HibernateDaoSupport/HibernateTemplate is not recommended since it unnecessarily ties your code to Spring classes.

Using these classes was inevitable with older versions of Hibernate in order to integrate support of Spring-managed transactions.

However, since Hibernate 3.0.1 you don't need it any more - you can write a code against a plain Hibernate API while using Spring-managed transactions. All you need is to configure Spring transaction support, inject SessionFactory and call getCurrentSession() on it when you need to work with session.

Another benefit of HibernateTemplate is exception translation. Without HibernateTemplate the same functionality can be achieved by using @Repository annotation, as shown in Gareth Davis's answer.

1. Ways to create queries in Hibernate

<http://www.javabeat.net/three-ways-to-create-query-in-hibernate/>

To create query in the ***Hibernate ORM framework***, there is three different types. The following are the three ways to create query instance:

1. *session.createQuery()*
2. *session.createSQLQuery()*
3. *session.createCriteria()*

## session.createQuery()

The method ***createQuery()*** creates ***Query*** object using the ***HQL*** syntax. For example

Query query = session.createQuery("from Student s where s.name

like 'k%'");

## session.createCriteria()

The method ***createCriteria()*** creates ***Criteria*** object for setting the query parameters.  
This is more useful feature for those who don’t want to write the query in hand. You can specify any type of complicated syntax using the ***Criteria API***.

Criteria criteria = session.createCriteria(Student.class);

Example

**package** javabeat.net.hibernate;

**import** java.util.List;

**import** org.hibernate.Criteria;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

/\*\*

\* source : www.javabeat.net

\*/

**public** **class** JavaBeatHibernateExample {

**public** **static** **void** main(String args[]) {

Configuration configuration = **new** Configuration();

// configuring hibernate

SessionFactory sessionFactory = configuration.configure().buildSessionFactory();

Session session = sessionFactory.openSession();

Criteria criteria = session.createCriteria(Student.**class**);

criteria.setMaxResults(10);

List<student> list = criteria.list();

**for** (Student student:list)

{

System.out.println(student.getName());

}

}

}

## Student.hbm.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>

<class name=*"javabeat.net.hibernate.Student"* table=*"****Student\_Details****"*>

<id name=*"id"* column=*"id"* type=*"java.lang.Long"*>

<generator class=*"increment"*/>

</id>

<property name=*"name"* column=*"name"* type=*"java.lang.String"*/>

<property name=*"dept"* column=*"dept"* type=*"java.lang.String"*/>

<property name=*"marks"* column=*"marks"* type=*"java.lang.Integer"*/>

<property name=*"section"* column=*"section"* type=*"java.lang.String"*/>

</class>

</hibernate-mapping>

Note: *“generator”:*

**[How to choose the id generation strategy when using hibernate](http://stackoverflow.com/questions/10041938/how-to-choose-the-id-generation-strategy-when-using-hibernate)**

All generators implement the interface org.hibernate.id.IdentifierGenerator. This is a very simple interface. Some applications can choose to provide their own specialized implementations, however, Hibernate provides a range of built-in implementations. The shortcut names for the built-in generators are as follows:

Around 12:

**increment**

Generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table. Do not use in a cluster.

**identity**

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

**sequence**

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**

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.

**seqhilo**

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

**uuid**

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

**guid**

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

**native**

selects identity, sequence or hilo depending upon the capabilities of the underlying database.

**assigned**

lets the application assign an identifier to the object before save() is called. This is the default strategy if no 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. It is usually used in conjunction with a primary key association.

**sequence-identity**

a specialized sequence generation strategy that utilizes a database sequence for the actual value generation, but combines this with JDBC3 getGeneratedKeys to return the generated identifier value as part of the insert statement execution. This strategy is only supported on Oracle 10g drivers targeted for JDK 1.4. Comments on these insert statements are disabled due to a bug in the Oracle drivers.

If you are building a simple application with not much concurrent users, you can go for **increment, identity, hilo** etc.. These are simple to configure and did not need much coding inside the db.

You should choose **sequence** or **guid** depending on your database. These are safe and better because the id generation will happen inside the database.

Example with annotations:

@Id

@GeneratedValue(strategy = *IDENTITY*)

@Column(name = "FACTOR\_ID", unique = **true**, nullable = **false**)

**public** Long getFactorId() {

**return** **this**.factorId;

}

## Native SQL Queries

The method ***createSQLQuery()*** creates ***Query*** object using the native ***SQL*** syntax. Fro example

Query query = session.createSQLQuery("Select \* from Student");

// In this example, you tell Hibernate to return you a Stock.class, all the select data (\*) will match to your Stock.class properties automatically.

Query query = session.createSQLQuery(

"select \* from stock s where s.stock\_code = :stockCode")

.addEntity(Stock.**class**)

.setParameter("stockCode", "7277");

List result = query.list();

// In this example, Hibernate will return you an Object array.

Query query = session.createSQLQuery(

"select s.stock\_code from stock s where s.stock\_code = :stockCode")

.setParameter("stockCode", "7277");

List result = query.list();

# Hibernate Named Query Examples

<http://www.mkyong.com/hibernate/hibernate-named-query-examples/>

Often times, developer like to put HQL string literals scatter all over the Java code, this method is hard to maintain and looks ugly. Fortunately, Hibernate come out a technique called “**names queries**” , It lets developer to put all HQL into the XML mapping file or via annotation.

Often times, developer like to put HQL string literals scatter all over the Java code, this method is hard to maintaine and look ugly. Fortunately, Hibernate come out a technique called “**names queries**” , it lets developer to put all HQL into the XML mapping file or via annotation.

**How to declare named query**

The named query is supported in both HQL or native SQL. see examples…

**1. XML mapping file**

HQL in mapping file

*<!-- stock.hbm.xml -->*

**<hibernate-mapping>**

**<class** name="com.mkyong.common.Stock" table="stock" ...**>**

**<id** name="stockId" type="java.lang.Integer"**>**

**<column** name="STOCK\_ID" **/>**

**<generator** class="identity" **/>**

**</id>**

**<property** name="stockCode" type="string"**>**

**<column** name="STOCK\_CODE" length="10" not-null="true" unique="true" **/>**

**</property>**

...

**</class>**

**<query** name="findStockByStockCode"**>**

<![CDATA[from Stock s where s.stockCode = :stockCode]]>

**</query>**

**</hibernate-mapping>**

Native SQL in mapping file

*<!-- stock.hbm.xml -->*

**<hibernate-mapping>**

**<class** name="com.mkyong.common.Stock" table="stock" ...**>**

**<id** name="stockId" type="java.lang.Integer"**>**

**<column** name="STOCK\_ID" **/>**

**<generator** class="identity" **/>**

**</id>**

**<property** name="stockCode" type="string"**>**

**<column** name="STOCK\_CODE" length="10" not-null="true" unique="true" **/>**

**</property>**

...

**</class>**

**<sql-query** name="findStockByStockCodeNativeSQL"**>**

**<return** alias="stock" class="com.mkyong.common.Stock"**/>**

<![CDATA[select \* from stock s where s.stock\_code = :stockCode]]>

**</sql-query>**

**</hibernate-mapping>**

You can place a named query inside ‘**hibernate-mapping**‘ element, but do not put before the ‘**class**‘ element, Hibernate will prompt invalid mapping file, all your named queries have to put after the ‘**class**‘ element.

**Note**  
Regarding the CDATA , it’s always good practice to wrap your query text with CDATA, so that the XML parser will not prompt error for some special XML characters like ‘>’ , <’ and etc.

**2. Annotation**

HQL in annotation

@NamedQueries({

@NamedQuery(

name = "findStockByStockCode",

query = "from Stock s where s.stockCode = :stockCode"

)

})

@Entity

@Table(name = "stock", catalog = "mkyong")

**public** **class** Stock **implements** java.io.Serializable {

...

Native SQL in annotation

@NamedNativeQueries({

@NamedNativeQuery(

name = "findStockByStockCodeNativeSQL",

query = "select \* from stock s where s.stock\_code = :stockCode",

resultClass = Stock.**class**

)

})

@Entity

@Table(name = "stock", catalog = "mkyong")

**public** **class** Stock **implements** java.io.Serializable {

...

In native SQL, you have to declare the ‘**resultClass**‘ to let Hibernate know what is the return type, failed to do it will caused the exception “**org.hibernate.cfg.NotYetImplementedException: Pure native scalar queries are not yet supported**“.

**Call a named query**

In Hibernate, you can call the named query via **getNamedQuery** method.

Query query = session.getNamedQuery("findStockByStockCode")

.setString("stockCode", "7277");

Query query = session.getNamedQuery("findStockByStockCodeNativeSQL")

.setString("stockCode", "7277");

**Conclusion**

Named queries are global access, which means the name of a query have to be unique in XML mapping files or annotations. In real environment, it’s always good practice to isolate all the named queries into their own file. In addition, named queries stored in the Hibernate mapping files or annotation are more easier to maintain than queries scattered through the Java code.

1. Use of [@MappedSuperclass](http://download.oracle.com/javaee/5/api/javax/persistence/MappedSuperclass.html).

<http://stackoverflow.com/questions/3509481/javax-persistence-annotations-and-inheritance?rq=1>

Designates a class whose mapping information is applied to the entities that inherit from it. A mapped superclass has no separate table defined for it.

A class designated with the MappedSuperclass annotation can be mapped in the same way as an entity except that the mappings will apply only to its subclasses since no table exists for the mapped superclass itself. When applied to the subclasses the inherited mappings will apply in the context of the subclass tables. Mapping information may be overridden in such subclasses by using the [AttributeOverride](http://docs.oracle.com/javaee/5/api/javax/persistence/AttributeOverride.html" \o "annotation in javax.persistence) and [AssociationOverride](http://docs.oracle.com/javaee/5/api/javax/persistence/AssociationOverride.html" \o "annotation in javax.persistence) annotations or corresponding XML elements.

Example

I have 4 persistent classes which all have the same fields (exactly) the only 3 difference between them is 1) the **class** name, 2) the table name and 3) the data. i am aware that **this** might seem strange to some but trust me there is a good reason which i won't go into here.

now, i'm using hibernate annotations to configure my class which should work like so:

@Entity

@Table(name = "store")

**public** **class** Store

{

@Id

@Column(name = "unique\_id")

**protected** String id;

@Column

**protected** String category;

...

}.. and **this** does work, **for** a single stand-alone **class**, however there are many fields to map and i'd like to do it all in one hit for all four similar classes, ie:

**public** **class** StoreBase

{

@Id

@Column(name = "unique\_id")

**protected** String id;

@Column

**protected** String category;

...

}

@Entity

@Table(name = "store1")

**public** **class** Store1 **extends** StoreBase

{}

@Entity

@Table(name = "store2")

**public** **class** Store2 **extends** StoreBase

{}

@Entity

@Table(name = "store3")

**public** **class** Store3 **extends** StoreBase

{}

@Entity

@Table(name = "store4")

**public** **class** Store4 **extends** StoreBase

{}however when attempting **this** i get the following exception: Caused by: org.hibernate.AnnotationException: No identifier specified for entity: package.entities.Store1

Ans: try

@MappedSuperclass

public class StoreBase

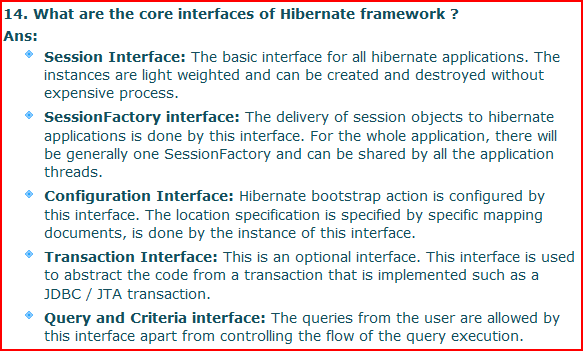
1. Hibernate mappings:

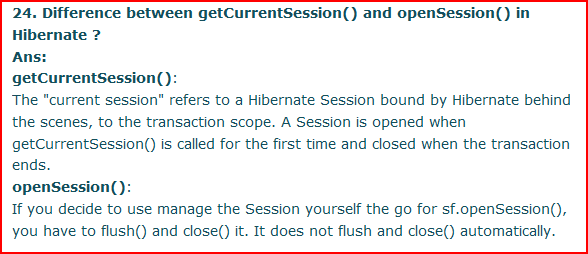
<http://www.mkyong.com/hibernate/hibernate-one-to-one-relationship-example/>

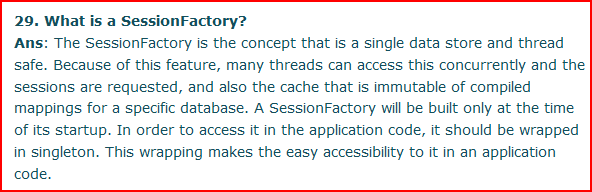
<http://www.mkyong.com/tutorials/hibernate-tutorials/>

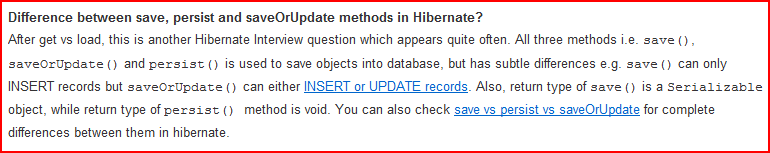
How to define table relationship **one-to-one, one-to-many, many-to-may** in Hibernate.

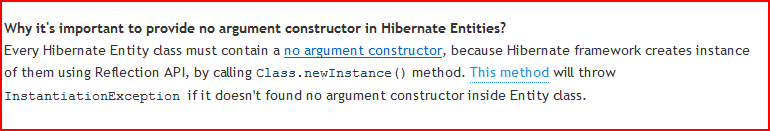
* [One-to-One example (XML Mapping)](http://www.mkyong.com/hibernate/hibernate-one-to-one-relationship-example/)  
  Hibernate one to one example with hbm mapping file.
* [One-to-One example (Annotation)](http://www.mkyong.com/hibernate/hibernate-one-to-one-relationship-example-annotation/)  
  Hibernate one to one example with annotation code.
* [One-to-Many example (XML Mapping)](http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/)  
  Hibernate one to many example with hbm mapping file.
* [One-to-Many example (Annotation)](http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example-annotation/)  
  Hibernate one to many example with annotation code.
* [Many-to-Many example (XML Mapping)](http://www.mkyong.com/hibernate/hibernate-many-to-many-relationship-example/)  
  Hibernate many to many example (NO extra column in join table) with hbm mapping file.
* [Many-to-Many example (Annotation)](http://www.mkyong.com/hibernate/hibernate-many-to-many-relationship-example-annotation/)  
  Hibernate many to many example (NO extra column in join table) with annotation code.
* [Many-to-Many example – join table + extra column (Annotation)](http://www.mkyong.com/hibernate/hibernate-many-to-many-example-join-table-extra-column-annotation/)  
  Hibernate many to many example (WITH extra column in join table) with annotation code.









****

**Different between session.get() and session.load()**

<http://www.mkyong.com/hibernate/different-between-session-get-and-session-load/>

**Difference between get and load in Hibernate?**

get vs load is one of the most frequently asked Hibernate Interview question, since correct understanding of both get() and load() is require to effectively using Hibernate. Main difference between get and load is that, **get will hit the database if object is not found in the cache and returned completely initialized object, which may involve several database call** while load() method can return proxy, if object is not found in cache and only hit database if any method other than getId() is called. This can save lot of performance in some cases. You can also see [difference between get and load in Hibernate](http://javarevisited.blogspot.com/2012/07/hibernate-get-and-load-difference-interview-question.html) for more differences and detailed discussion on this question.

1. get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.
2. Since load() throws exception when data is not found, we should use it only when we know data exists.
3. We should use get() when we want to make sure data exists in the database.

That’s all for hibernate get and load methods, I hope it will clear some doubts and help you in deciding which one to use in different scenarios.

**1. Behavior when Object is not found in Session Cache**

Apart from performance this is another difference between get and load which is worth remembering. get method of Hibernate Session class returns null if object is not found in cache as well as on database while load() method [throws](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) ObjectNotFoundException if object is not found on cache as well as on database but never return null.

**2. Database hit**

Get method always hit [database](http://javarevisited.blogspot.sg/2011/10/how-to-use-truncate-and-delete-command.html) while load() method may not always hit the database, depending upon which method is called.

**3. Proxy**

Get method never returns a proxy, it either returns null or fully initialized Object, while load() method may return proxy, which is the object with ID but without initializing other properties, which is lazily initialized. If you are just using returned object for creating relationship and only need Id then load() is the way to go.

**4. Performance**

By far most important difference between get and load in my opinion. get method will return a completely initialized object if  Object is not on the cache but exists on [Database](http://javarevisited.blogspot.sg/2011/10/selct-command-sql-query-example.html), which may involve multiple round-trips to database based upon object relational mappings while load() method of Hibernate can return a **proxy** which can be initialized on demand (lazy initialization) when a non identifier method is accessed. Due to above reason use of load method will result in slightly **better performance**, but there is a caveat that proxy object will throw ObjectNotFoundException later if corresponding row doesn’t exists in database, instead of failing immediately so not a [fail fast](http://javarevisited.blogspot.sg/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) behavior.

5. load method exists prior to get method which is added on user request.

# [What is proxy in the context of load() method of Hibernate?](http://stackoverflow.com/questions/20988626/what-is-proxy-in-the-context-of-load-method-of-hibernate)

Proxies are classes dynamically by Hibernate generated to help lazy loading. For instance if you have a Catclass, Hibernate will generate a proxy class that extends Cat.

If you get an uninitialized instance of this proxy, essentially all its fields will be null except the ID because Hibernate has not yet hit the database. Now the first time you will call a method on this proxy, it will realize that it is not initialized and it will query the database to load it's attributes. This is possible because the dynamically generated class overrides the base class's methods and adds this initialized/uninitialized check.

Now assume that your Cat class is not a proxy and that it has a father association, when you load a Cat object, Hibernate will need to load all it's attributes. So if you load a Cat object, Hibernate will also need to load its father and the father's father and so on. Using proxies enable Hibernate to only load the required instances.

Cat cat1 = (Cat) session.load(1);

Cat cat2 = (Cat) session.load(2);

Cat cat3 = (Cat) session.load(3);

cat1.meow(); // this will cause Hibernate to run a query to load cat1's data

cat2.meow(); // this will cause Hibernate to run a query to load cat2's data

// After this cat3 is still an uninitiated proxy because it has not been used

batch-size is another feature of Hibernate that, in most instances, help dealing with lazy loading. Basically the idea is that Hibernate keeps track of the uninitialized proxies and when one of the needs to be initialized, a single query will be executed to load *up to* batch-size proxies (instead of just one proxy/query)

Cat cat1 = (Cat) session.load(1);

Cat cat2 = (Cat) session.load(2);

cat1.meow(); // if batch-size >= 2, cat1 and cat2 will be loaded in a single query

cat2.meow(); // no query will be executed here

best example:

<http://www.mkyong.com/hibernate/different-between-session-get-and-session-load/>

Read more: <http://javarevisited.blogspot.com/2013/05/10-hibernate-interview-questions-answers-java-j2ee-senior.html#ixzz39LiLiGch>

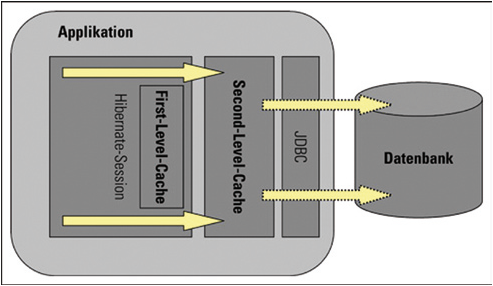
<http://howtodoinjava.com/2013/07/02/how-hibernate-second-level-cache-works/>

**HOW SECOND LEVEL CACHE WORKS**

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.

* **Fist level cache**: This is enabled by default and works in **session** scope. Read more about [**hibernate first level cache**](http://howtodoinjava.com/2013/07/01/understanding-hibernate-first-level-cache-with-example/).
* **Second level cache**: This is apart from first level cache which is available to be used globally in session factory scope.

Above statement means, **second level cache is created in session factory scope** and is **available to be used in all sessions** which are created using that particular session factory.  
It also means that **once session factory is closed, all cache associated with it die** and cache manager also closed down.  
Further, It also means that if you have two instances of session factory (normally no application does that), you will have two cache managers in your application and while accessing cache stored in physical store, you might get unpredictable results like cache-miss.



*Hibernate first and second level cache*

In this tutorial, I am giving concepts around hibernate second level cache and give example using code snippets.

### ****How second level cache works****

Lets write all the facts point by point:

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | /\*\*   \* Evicts all second level cache hibernate entites. This is generally only   \* needed when an external application modifies the databaase.   \*/  public void evict2ndLevelCache() {      try {          Map<String, ClassMetadata> classesMetadata = sessionFactory.getAllClassMetadata();          for (String entityName : classesMetadata.keySet()) {              logger.info("Evicting Entity from 2nd level cache: " + entityName);              sessionFactory.evictEntity(entityName);          }      } catch (Exception e) {          logger.logp(Level.SEVERE, "SessionController", "evict2ndLevelCache", "Error evicting 2nd level hibernate cache entities: ", e);      }  } |

To understand more using examples, I wrote an application for testing in which I configured EhCache as 2nd level cache. Lets see various scenarios:

**a) Entity is fetched very first time**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 0    Output: 1 0 |

***Explanation***: Entity is not present in either 1st or 2nd level cache so, it is fetched from database.

**b) Entity is fetched second time**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 0    Output: 1 0 |

***Explanation***: Entity is present in first level cache so, it is fetched from there. No need to go to second level cache.

**c) Entity is evicted from first level cache and fetched again**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //Evict from first level cache  session.evict(department);    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 1    Output: 1 1 |

***Explanation***: First time entity is fetched from database. Which cause it store in 1st and 2nd level cache. Second load call fetched from first level cache. Then we evicted entity from 1st level cache. So third load() call goes to second level cache and getSecondLevelCacheHitCount() returns 1.

**d) Access second level cache from another session**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //Evict from first level cache  session.evict(department);    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 2    Output: 1 2 |

***Explanation***: When another session created from same session factory try to get entity, it is successfully looked up in second level cache and no database call is made.

So now we are clear on how second level cache is used by hibernate.

Lazy initialization:

<http://stackoverflow.com/questions/978759/what-is-lazy-initialization-and-why-is-it-useful>

# [Difference between FetchType LAZY and EAGER in Java persistence?](http://stackoverflow.com/questions/2990799/difference-between-fetchtype-lazy-and-eager-in-java-persistence)

Sometimes you have two entities and there's a relationship between them. For example, you might have an entity called University and another entity called Student.

The University entity might have some basic properties such as id, name, address, etc. as well as a property called students:

public class University {

private String id;

private String name;

private String address;

private List<Student> students;

// setters and getters

}

Now when you load a University from the database, JPA loads its id, name, and address fields for you. But you have two options for students: to load it together with the rest of the fields (i.e. eagerly) or to load it on-demand (i.e. lazily) when you call the university's getStudents() method.

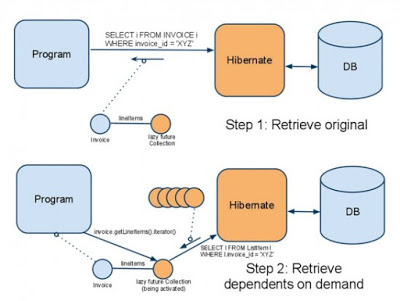
When a university has many students it is not efficient to load all of its students with it when they are not needed. So in suchlike cases, you can declare that you want students to be loaded when they are actually needed. This is called lazy loading.

EAGER loading of collections means that they are fetched fully at the time their parent is fetched. So if you have Course and it has List<Student>, all the students are fetched *from the database* at the time the Course is fetched.

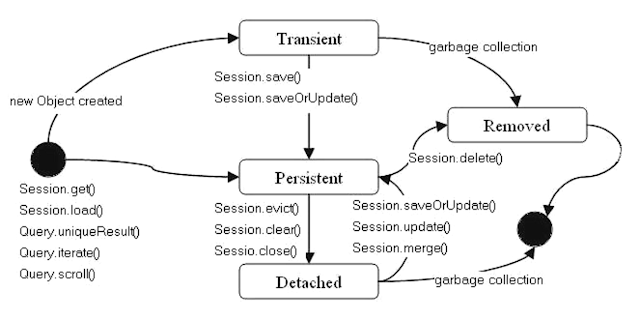
LAZY on the other hand means that the contents of the List are fetched only when you try to access them. For example, by calling course.getStudents().iterator(). Calling any access method on the List will initiate a call to the database to retrieve the elements. This is implemented by creating a Proxy around the List(or Set). So for your lazy collections, the concrete types are not ArrayList and HashSet, but PersistentSet and PersistentList (or PersistentBag)

## 20 Hibernate Questions from Java JEE Interviews

Here is my selected list of 20 Hibernate based questions for Java developers. It contains questions from Hibernate fundamentals, one-to-one, and one-to-many mappings, caching, Hibernate vs JDBC comparison, pros and cons of Hibernate, known problems with Hibernate, and Performance improvement.  
  
  
**1. What is Hibernate?**  
Hibernate is an ORM (Object-relational Mapping) framework, which allows the developer to concentrate on business logic by taking care of the persistence of data by itself. Java developer can write code using object and Hibernate can take care of creating those object from data loaded from the database and saving update back to the database.  
  
  
**2. What are the advantages of Hibernate over JDBC?**(detailed answer)  
Apart from Persistence i.e. saving and loading data from Database, Hibernate also provides following benefits  
1) Caching  
2) Lazy Loading  
3) Relationship management and provides code for mapping an object to the data  
4) The developer is free from writing code to load/store data into the database.  
  
  
**3. Difference between get() vs load() method in Hibernate?** ([detailed answer](http://javarevisited.blogspot.com/2012/07/hibernate-get-and-load-difference-interview-question.html))  
This is one of the most frequently asked Hibernate interview questions, I have seen it several times. The key difference between the get() and load() method is that load() will throw an exception if an object with id passed to them is not found, but get() will return null.  
  
Another important difference is that load can return proxy without hitting the database unless required (when you access any attribute other than id) but get() always go to the database, so sometimes using load() can be faster than the get() method.  
  
Use the load() method, if you know the object exists, and get() method if you are not sure about the object's existence. See [Introduction To Hibernate](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fhibernate-introduction) course on Pluralsight to learn more about how Hibernate works

[](https://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Fcourses%2Fspring-jpa-hibernate)

Btw, you would need a Pluralsight membership to access this course, which costs around $29 monthly or $299 annually. I have one and I also suggest all developers have that plan because Pluralsight is like NetFlix for Software developers.  
  
It has more than 5000+ good quality courses on all latest topics. Since we programmers have to learn new things every day, an investment of $299 USD is not bad.  
  
Btw, it also offers a [10-day free trial](http://pluralsight.pxf.io/c/1193463/424552/7490?u=https%3A%2F%2Fwww.pluralsight.com%2Flearn) without any obligation which allows you to watch 200 hours of content. You can watch this course for free by signing for that trial.  
  
  
  
**4. What is N+1 SELECT problem in Hibernate?**(detailed answer)  
The N+1 SELECT problem is a result of lazy loading and load on demand fetching strategy. In this case, Hibernate ends up executing N+1 SQL queries to populate a collection of N elements.  
  
For example, if you have a List of N Items where each Item has a dependency on a collection of Bid object. Now if you want to find the highest bid for each item then Hibernate will fire 1 query to load all items and N subsequent queries to load Bid for each item.  
  
So in order to find the highest bid for each item your application end up firing N+1 queries.  It's one of the important Hibernate interview questions and I suggest to read chapter 13 of [Java Persistence with Hibernate](http://www.amazon.com/Java-Persistence-Hibernate-Christian-Bauer/dp/1932394885?tag=javamysqlanta-20) to understand this problem in more details.  
  
  
 **5. What are some strategies to solve the N+1 SELECT problem in Hibernate?**(detailed answer)  
This is the follow-up question of previous Hibernate interview question. If you answer the last query correctly then you would be most likely asked this one.  
  
Here are some strategies to solve the N+1 problem:  
1) pre-fetching in batches, this will reduce the N+1 problem to N/K + 1 problem where  K is the size of the batch  
2) subselect fetching strategy  
3) disabling lazy loading  
  
  
 **5. What is the difference between save() and persist() method in Hibernate?**([detailed answer](http://java67.blogspot.com/2016/01/difference-between-save-saveorupdate-and-persist-in-Hibernate.html))  
The main difference between save() and persist() method is that, save returns a Serializable object while return type of the persist() method is void, so it doesn't return anything.  
  
Here is a nice diagram which explains the state transition in Hibernate:

[](https://javarevisited.blogspot.com/2018/01/top-5-hibernate-and-jpa-courses-for-java-programmers-learn-online.html)

**7. What is the requirement for a Java object to become a Hibernate entity object?**([detailed answer](http://javarevisited.blogspot.com/2016/01/why-jpa-entity-or-hibernate-persistence-should-not-be-final-in-java.html))  
It should not be final and must provide a default, no-argument constructor. See the detailed answer to learn more about the special requirement for a Java object to become a Hibernate Entity.  
  
  
 **8. What are different types of caches available in Hibernate?**(detailed answer)  
This is another common Hibernate interview question. Hibernate provides the out-of-box caching solution but there are many caches e.g. first level cache, second level cache, and query cache.  
  
The first level cache is maintained at Session level and cannot be disabled but the second level cache is required to be configured with external cache provider like EhCache.  
  
  
**9. What is the difference between the first and second level cache in Hibernate?**(detailed answer)  
This is again follow-up of previous Hibernate interview question. The first level cache is maintained at Session level while the second level cache is maintained at a SessionFactory level and shared by all sessions. You can read [these books](http://javarevisited.blogspot.com/2014/01/top-5-hibernate-books-for-java-programmers-learning.html) to learn more about caching in Hibernate.  
  
  
**10. Does Hibernate Session interface thread-safe in Java?**(detailed answer)  
No, Session object is not thread-safe in Hibernate and intended to be used with-in single thread in the application.  
  
  
 **11. Does SessionFactory thread-safe in Hibernate?**([detailed answer](http://javarevisited.blogspot.com/2013/05/10-hibernate-interview-questions-answers-java-j2ee-senior.html))  
SessionFactory is both Immutable and thread-safe and it has just one single instance in Hibernate application. It is used to create a Session object and it also provides caching by storing SQL queries stored by multiple session.  
  
The second level cache is maintained at SessionFactory level. This can be a difficult and tricky question for less experienced Java developers who are not familiar with [thread-safety](http://www.java67.com/2015/09/thread-safe-singleton-in-java-using-double-checked-locking-pattern.html) and [Immutability](http://javarevisited.blogspot.sg/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html).  
  
  
**12. What is different between Session and SessionFactory in Hibernate?**(detailed answer)  
This is another popular Hibernate interview question, mostly at a telephonic round of interviews.  
  
The main difference between Session and SessionFactory is that former is a single-threaded, short-lived object while later is Immutable and shared by all Session.  
  
It also lives until the Hibernate is running. Another difference between Session and SessionFactory is that former provides first level cache while SessionFactory provides the Second level cache.  
  
  
**13. What is criterion query in hibernate?**(detailed answer)  
Criteria is a simplified API for retrieving entities by composing Criterion objects also known as Criterion query.  
  
This is a very convenient approach for functionality like "search" screens where you can filter data on multiple conditions as shown in the following example:

List books = session.createCriteria(Book.class)

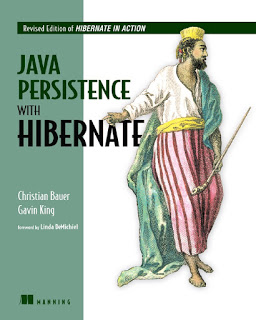
.add(Restrictions.like("name", "java%") )

.add(Restrictions.like("published\_year", "2015"))

.addOrder(Order.asc("name") )

.list();

This can be a tough question if you are not using Hibernate on a daily basis, I have interviewed several Java developers who have used Hibernate but doesn't know about Criterion query or API. You can check out this list of [5 Hibernate and JPA courses for Java Programmers](https://javarevisited.blogspot.com/2018/01/top-5-hibernate-and-jpa-courses-for-java-programmers-learn-online.html) to learn more about JPA and Hibernate.  
  
  
**14. What are other ORM frameworks? Any alternative of Hibernate?**  
This is a general question, sometimes asked to start the conversation and other times to finish the interview. EJB and TopLink from Oracle are two of the most popular alternative to Hibernate framework.  
  
  
**15. What is the difference between save() and saveOrUpdate() method of Hibernate?**([detailed answer](http://javarevisited.blogspot.com/2012/09/difference-hibernate-save-vs-persist-and-saveOrUpdate.html))  
Though both save() and saveOrUpdate() method is used to store an object into Database, the key difference between them is that save can only INSERT records but saveOrUpdate() can either INSERT or UPDATE records.  
  
  
**16. What is difference between getCurrentSession() and openSession() in Hibernate?**(detailed answer)  
SessionFactory.openSession() always opens a new session that you have to close once you are done with the operations. SessionFactory.getCurrentSession() returns a session bound to a context - you don't need to close this.  
  
  
**17. What is Hibernate Query Language (HQL)?**([detailed answer](http://javarevisited.blogspot.com/2014/03/hibernate-nullpointerexception-due-to-space-in-HQL-Query.html))  
Hibernate query language, HQL is an object-oriented extension to SQL. It allows you to query, store, update, and retrieve objects from a database without using SQL.  
  
This question is also similar to the earlier question about Criterion query, Java developers who have not used Hibernate extensively will not know much about features like HQL and Criterion.  
  
  
**18. When do you use merge() and update() in Hibernate?**(detailed answer)  
This is one of the tricky Hibernate interview questions. You should use update() if you are sure that the Hibernate session does not contain an already persistent instance with the same id and use merge() if you want to merge your modifications at any time without considering the state of the session. See [Java Persistence with Hibernate](http://www.amazon.com/Java-Persistence-Hibernate-Christian-Bauer/dp/1932394885?tag=javamysqlanta-20) for more details.

[](https://javarevisited.blogspot.com/2018/08/top-5-hibernate-books-for-java.html)

**19. The difference between sorted and ordered collection in Hibernate?**(detailed answer)  
The main difference between sorted and ordered collection is that sorted collection sort the data in JVM's heap memory using Java's collection framework sorting methods while the ordered collection is sorted using order by clause in the database itself.  
  
A sorted collection is more suited for small dataset but for a large dataset, it's better to use ordered collection to avoid [OutOfMemoryError in Java](http://java67.blogspot.com/2013/08/guide-of-javalangoutofmemoryerror-java-heap-space-tomcat-eclipse-minecraft-jboss.html) application.  
  
  
**20. How do you log SQL queries issued by the Hibernate framework in Java application?**  
You can use the show\_sql property to log SQL queries issued by the Hibernate framework, Just add the following line in your Hibernate configuration file:

<property name=”show\_sql”> true </property>

**21. What are the three states of a Hibernate Persistence object can be?**(detailed answer)  
The Hibernate persistent or entity object can live in the following three states:  
1) transient

An object which is not associated with hibernate session and does not represent a row in the database is considered as transient.  
2) persistent

Employee emp = new Employee();

emp.setName("Ravi Raj");

Long id = (Long) session.save(emp);

// emp object is now in a persistent state

An object that is associated with the hibernate session is called as Persistent object.

3) detached

session.close();

Object which is just removed from hibernate session is called as detached object.The sate of the detached object is called as detached state.

<http://javawebtutor.com/articles/hibernate/hibernate-object-states.php>

**22. What is the difference between the transient, persistent and detached state in Hibernate?**(detailed answer)  
New objects created in Java program but not associated with any hibernate Session are said to be in the transient state.  
  
On the other hand, an object which is associated with a Hibernate session is called Persistent object. While an object which was earlier associated with Hibernate session but currently it's not associate is known as a detached object.  
  
You can call save() or persist() method to store those object into the database and bring them into the Persistent state. Similarly, you can re-attach a detached object to hibernate sessions by calling either update() or saveOrUpdate() method. See [Spring and Hibernate for Beginners](https://click.linksynergy.com/link?id=JVFxdTr9V80&offerid=323058.647428&type=2&murl=https%3A%2F%2Fwww.udemy.com%2Fspring-hibernate-tutorial%252) to learn more about persistence object's lifecycle in Hibernate.  
  
  
**23. Which cache is used by Session Object in Hibernate? First level or second level cache?**(detailed answer)  
A Session object uses the first-level cache. As I told before the second level cache is used at SessionFactory level. This is a good question to check if Candidate has been working in hibernate or not. If he has not worked in Hibernate from a long time then he would get confused in this question.  
  
  
That's all in this list of **Hibernate Interview question for Java and JEE developers**. In this article, we have covered a lot of frequently asked Hibernate question for both beginners and experienced Java developers from all important topics of Hibernate framework e.g. Hibernate fundamentals, caching, collection mapping, performance tuning, common issues and Hibernate vs JDBC.  
  
If you find that you lack knowledge in any particular area, I suggest go through following resources to fill those gaps in your knowledge.  
  
Read more: <http://www.java67.com/2016/02/top-20-hibernate-interview-questions.html#ixzz5eocg4ZcG>

# How to maintain history of tables in Hibernate

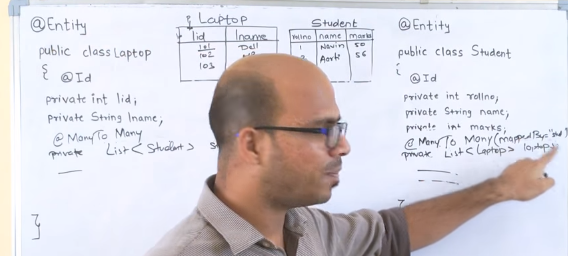
To Maintain history of database or keep track of modification of database table row , we create a version table that contains fields identical to original table.Whenever original table gets changed , we create another entry in version table. So for every update query , we have to write an insert query in version table . There is a module available in hibernate to manage easy auditing of objects , and we don’t have to write separate insert query by ourselves.

Hibernate Envers provides in built mechanism to maintain history of objects in database . [Envers](http://www.jboss.org/envers)is library for Hibernate that will help us to easily achieve audit functionality. This has been created by [Adam Warski](http://www.warski.org/blog/). From Hibernate 3.5, Envers is included as a Hibernate core module. Let’s take an example as to how Envers can be used to maintain history of objects.

Hibernate supports the following mapping strategies:

* **Single table per class**: Both superclass and subclass are mapped to the same table. An additional column marks whether the row is an instance of the superclass or subclass and fields that are not present in the superclass are left empty.
* **Joined subclass**: This strategy uses a separate table for each class whereas the table for the subclass only stores the fields that are not present in the superclass. To retrieve all values for an instance of the subclass, a join between the two tables has to be performed.
* **Table per class**: This strategy also uses a separate table for each class but stores in the table for the subclass also the fields of the superclass. With this strategy one row in the subclass table contains all values and in order to retrieve all values no join statement is necessary.

Hibernate relationships



# [What is the difference between Get and Load functions in Hibernate](https://stackoverflow.com/questions/20799684/what-is-the-difference-between-get-and-load-functions-in-hibernate)

load() will return a proxy of the row or object in the database. It will throw an ObjectNotFoundException. It will never return null.

get() represents an actual row in the database, so it is not a proxy and will not throw an exception if not found, but will return null.

## Difference between save and persist method in Hibernate

In last section we saw What are difference between save and saveOrUpdate and now we will see Difference on save vs persist method.

1)First difference between save and persist is there return type. Similar to save method persist also INSERT records into database but **return type of persist is void** while return type of save is [Serializable](http://javarevisited.blogspot.sg/2012/01/serializable-externalizable-in-java.html) object.

2) Another difference between persist and save is that both methods make a [transient](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html) instance persistent. However, persist () method doesn't guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time.

3) Fourth difference between save and persist method in Hibernate is related to previous difference on save vs persist. Because of its above behavior of persist method outside transaction boundary, its useful in long-running conversations with an extended Session context. On the other hand save method is not good in a long-running conversation with an extended Session context.  
  
Read more: <https://javarevisited.blogspot.com/2012/09/difference-hibernate-save-vs-persist-and-saveOrUpdate.html#ixzz5fAvInGZt>

Read more: <https://javarevisited.blogspot.com/2012/09/difference-hibernate-save-vs-persist-and-saveOrUpdate.html#ixzz5fAuVUVeP>