Spring With HIBERNATE

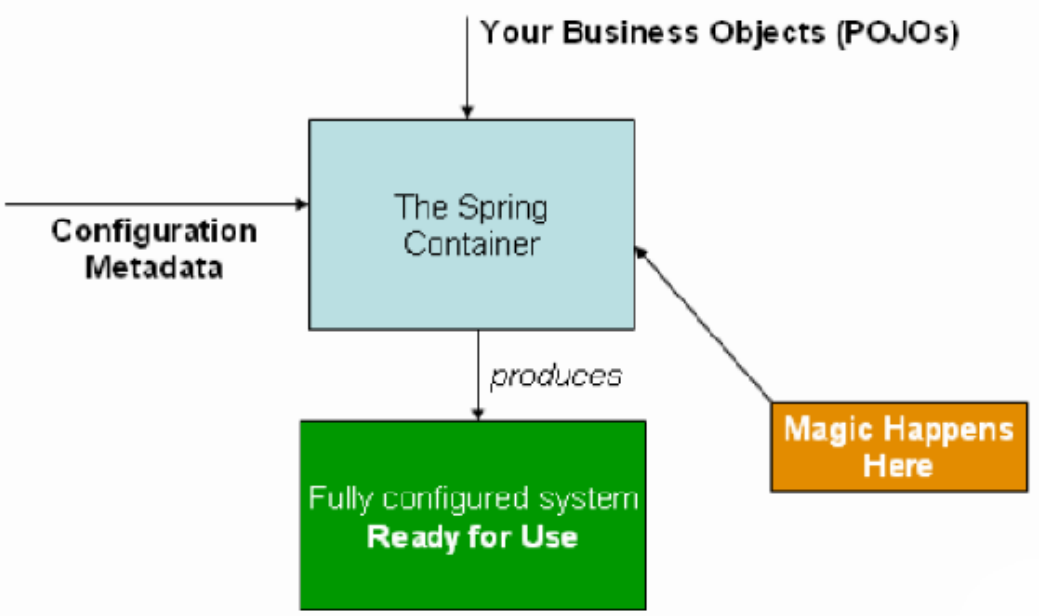
SPRING

IOC(Inversion of Control)

The **org.springframework.context.ApplicationContext** is the actual representation of the Spring IoC container that is responsible for containing and otherwise managing the aforementioned beans.

The **ApplicationContext** interface is the central IOC container interface in Spring. Its responsibilities include instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects.

ClassPathXmlApplicationContext is a subclass of ApplicationContext. This implementation allows you to express the objects that compose your application, and the doubtless rich interdependencies between such objects, in terms of XML. The ClassPathXmlApplicationContext takes this XML configuration metadata and uses it to create a fully configured system or application.



A Spring IOC container manages one or more beans. These beans are created using the instructions defined in the configuration metadata that has been supplied to the container (typically in the form of XML <bean/> definitions).

Withinthe container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:

- **a package-qualified class name:** this is normally the actual implementation class of the bean being defined. However, if the bean is to be instantiated by invoking a static factory method instead of using a normal constructor, this will actually be the class name of the factory class.
- bean behavioral configuration elements, which state how the bean should behave in the container (i.e. prototype or singleton, autowiring mode, dependency checking mode, initialization and destruction methods)

- Constructor arguments and property values to set in the newly created bean. An example would be the number of connections to use in a bean that manages a connection pool (either specified as a property or as a constructor argument), or the pool size limit.
- Other beans which are needed for the bean to do its work, i.e. collaborators (also called dependencies).

Instantiation using a constructor

When creating a bean using the constructor approach, all normal classes are usable by and compatible with spring. That is, the class being created does not need to implement any specific interfaces or be coded in a specific fashion. Just specifying the bean class should be enough. However, depending on what type of IOC you are going to use for that specific bean, you may need a default (empty) constructor.

Additionally, the Spring IOC container isn't limited to just managing true JavaBeans, it is also able to manage virtually any class you want it to manage. Most people using spring prefer to have actual JavaBeans (having just a default (no-argument) constructor and appropriate setters and getters modeled after the properties) in the container, but it is also possible to have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, spring can manage it as well.

When using XML-based configuration metadata you can specify your bean class like so:

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

The mechanism for supplying arguments to the constructor (if required), or setting properties of the object instance after it has been constructed, will be described shortly.

Instantiation using a static factory method

When defining a bean which is to be created using a static factory method, along with the class attribute which specifies the class containing the static factory method, another attribute named factory-method is needed to specify the name of the factory method itself. Spring expects to be able to call this method (with an optional list of arguments as described later) and get back a live object, which from that point on is treated as if it had been created normally via a constructor. One use for such a bean definition is to call static factories in legacy code.

The following example shows a bean definition which specifies that the bean is to be created by calling a factory-method. Note that the definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the createInstance () method must be a static method.

```
<bean id="exampleBean"
Class="examples.ExampleBean2"
Factory-method="createInstance"/>
```

The mechanism for supplying (optional) arguments to the factory method, or setting properties of the object instance after it has been returned from the factory, will be described shortly.

Instantiation using an instance factory method

In a fashion similar to instantiation via a static factory method, instantiation using an instance factory method is where the factory method of an existing bean from the container is invoked to create the new bean.

To use this mechanism, the 'class' attribute must be left empty, and the 'factory-bean' attribute must specify the name of a bean in the current (or parent/ancestor) container that contains the factory method. The factory method itself must still be set via the 'factory-method' attribute (as seen in the example below).

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="myFactoryBean" class="...">
...
</bean>

<! -- the bean to be created via the factory bean -->
<bean id="exampleBean"
Factory-bean="myFactoryBean"
Factory-method="createInstance"/>
```

Although the mechanisms for setting bean properties are still to be discussed, one implication of this approach is that the factory bean itself can be managed and configured via DI.

Autowiring collaborators

A Spring IOC container is able to autowire relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the ApplicationContext. The autowiring functionality has five modes. Autowiring is specified per bean and can thus be enabled for some beans, while other beans won't be autowired. Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor arguments, saving a significant amount of typing. When using XML-based configuration metadata, the autowire mode for a bean definition is specified by using the autowire attribute of the <bean/> element.

The following values are allowed:

Autowiring modes

Mode	Explanation

no	No autowiring at all. Bean references must be defined via a ref element. This is the default, and changing this is discouraged for larger deployments, since explicitly specifying collaborators gives greater control and clarity. To some extent, it is a form of documentation about the structure of a system.
byName	Autowiring by property name. This option will inspect the container and look for a bean named exactly the same as the property which needs to be autowired. For example, if you have a bean definition which is set to autowire by name, and it contains a master property (that is, it has a setMaster (...) method), Spring will look for a bean definition named master, and use it to set the property.
byType	Allows a property to be autowired if there is exactly one bean of the property type in the container. If there is more than one, a fatal exception is thrown, and this indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. If this is not desirable, setting the dependency-check="objects" attribute value specifies that an error should be thrown in this case.
constructor	This is analogous to byType, but applies to constructor arguments. If there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised.
autodetect	Chooses constructor or byType through introspection of the bean class. If a default constructor is found, the byType mode will be applied.

Note that explicit dependencies in property and constructor-arg settings always override autowiring. Please also note that it is not currently possible to autowire socalled simple properties such as primitives, Strings, and Classes (and arrays of such simple properties).(This is by-design and should be considered a feature.) Autowire behavior can be combined with dependency checking, which will be performed after all autowiring has been completed.

It is important to understand the various advantages and disadvantages of autowiring. Some advantages of autowiring include:

- Autowiring can significantly reduce the volume of configuration required. However, mechanisms such as the use of a bean template are also valuable in this regard.
- Autowiring can cause configuration to keep itself up to date as your objects evolve.
  For example, if you need to add an additional dependency to a class, that dependency can be satisfied automatically without the need to modify configuration. Thus there may be a strong case for autowiring during development, without ruling out the option of switching to explicit wiring when the code base becomes more stable.

Some disadvantages of autowiring:

- Autowiring is more magical than explicit wiring. Although, as noted in the above table, spring is careful to avoid guessing in case of ambiguity which might have unexpected results, the relationships between your Spring-managed objects is no longer explicitly documented.
- Wiring information may not be available to tools that may generate documentation from a spring container.
- Autowiring by type will only work when there is a single bean definition of the type specified by the setter method or constructor argument. You need to use explicit wiring if there is any potential ambiguity.

There is no "wrong" or "right" answer in all cases. A degree of consistency across a project is best though; for example, if autowiring is not used in general, it might be confusing to developers to use it just to wire one or two bean definitions.

Excluding a bean from being available for autowiring

You can also (on a per bean basis) totally exclude a bean from being an autowire candidate. When configuring beans using Spring's XML format, the '**autowirecandidate**' attribute of the **<bean/>** element can be set to '**false**'; this has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

This can be useful when you have a bean that you absolutely never ever want to have injected into other beans via autowiring. It does not mean that the excluded bean cannot itself be configured using autowiring... it can, it is rather that it itself will not be considered as a candidate for autowiring other beans.

Checking for dependencies

The Spring IOC container also has the ability to try to check for the existence of unresolved dependencies of a bean deployed into the container. These are JavaBeans properties of the bean, which do not have actual values set for them in the bean definition, or alternately provided automatically by the autowiring feature.

This feature is sometimes useful when you want to ensure that all properties (or all properties of a certain type) are set on a bean. Of course, in many cases a bean class will have default values for many properties, or some properties do not apply to all usage scenarios, so this feature is of limited use. Dependency checking can also be enabled and disabled per bean, just as with the autowiring functionality. The default is to not check dependencies. Dependency checking can be handled in several different modes. When using XML-based configuration metadata, this is specified via the '**dependency-check**' attribute in a bean definition, which may have the following values.

Dependency checking modes

Mode	Explanation

none	No dependency checking. Properties of the bean which have no value specified for them are simply not set.
simple	Dependency checking is performed for primitive types and collections (everything except collaborators, i.e. other beans)
object	Dependency checking is performed for collaborators only
all	Dependency checking is done for collaborators, primitive types and collections

Bean scopes

When you create a bean definition (typically in an XML configuration file) what you are actually creating is (loosely speaking) a recipe or template for creating actual instances of the objects defined by that bean definition. The fact that a bean definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the scope of the objects created from a particular bean definition. This approach is very powerful and gives you the flexibility to choose the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware Spring ApplicationContext).

The scopes supported out of the box are listed below:

Bean scopes

Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; i.e. each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

The singleton scope

When a bean is a singleton, only one shared instance of the bean will be managed and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create exactly one instance of the object defined by that bean definition (or recipe). This single instance will be stored in a singleton cache, and all subsequent requests and references for that named bean will result in the cached object instance being returned.

The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you would write configuration like so:

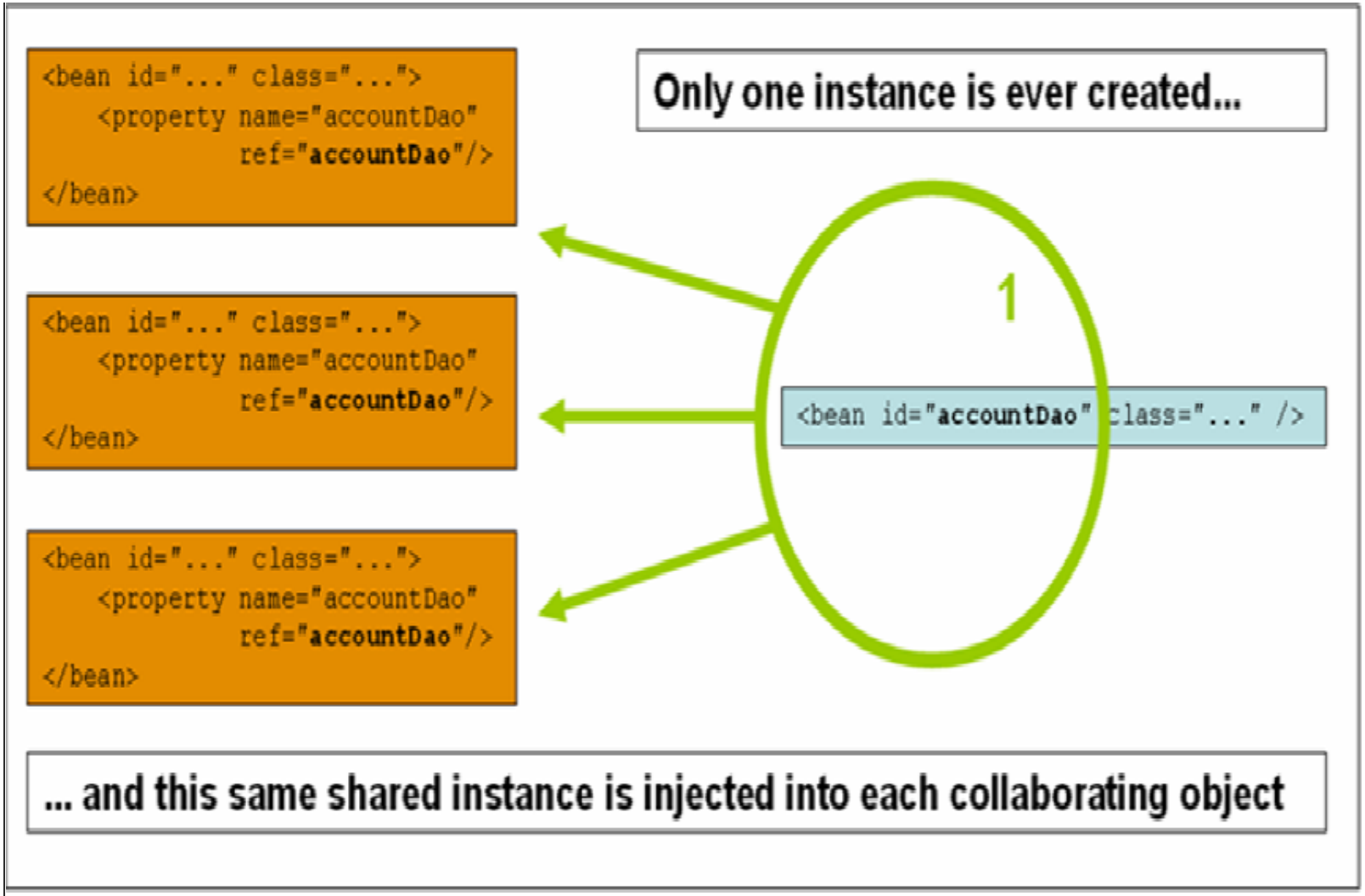
```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<! -- The following is equivalent, though redundant (singleton scope is the default); using
spring-beans-2.0.dtd -->
```

```
<bean id="accountService" class="com.foo.DefaultAccountService"
scope="singleton"/>

<! -- The following is equivalent and preserved for backward compatibility in springbeans.
dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService"
singleton="true"/>
```

The following diagram illustrates the Spring singleton scope.



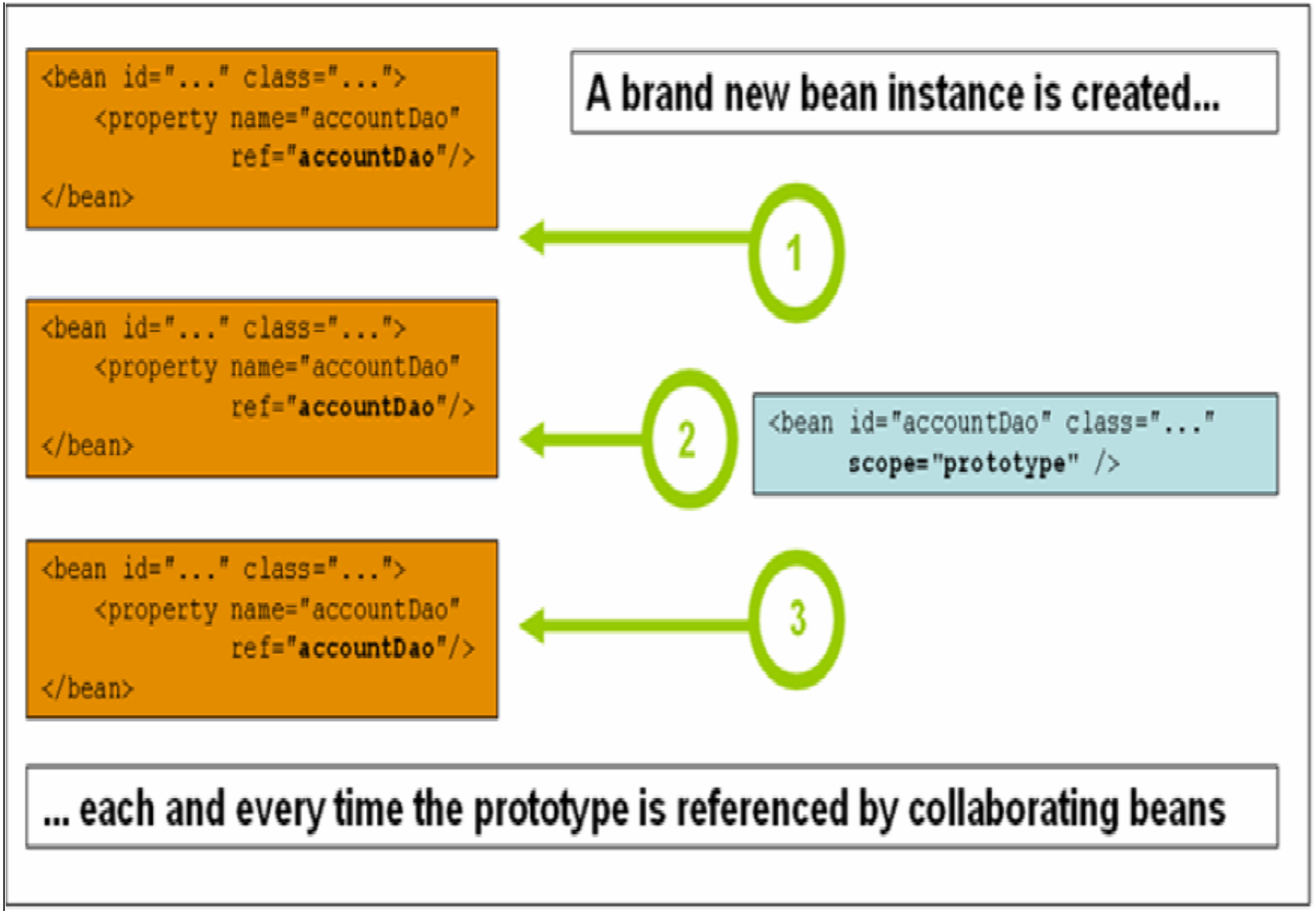
Please be aware that spring's concept of a singleton bean is quite different from the Singleton pattern as defined in the seminal Gang of Four (GoF) patterns book. The classic GoF Singleton hardcodes the scope of an object such that one and only one instance of a particular class will ever be created per ClassLoader. The scope of the Spring singleton is best described as per container and per bean. This means that if you define

one bean for a particular class in a single Spring container, then the Spring container will create one and only one instance of the class defined by that bean definition.

The prototype scope

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made (that is, it is injected into another bean or it is requested via a programmatic getBean() method call on the container). As a rule of thumb, you should use the prototype scope for all beans that are stateful, while the singleton scope should be used for stateless beans.

The following diagram illustrates the Spring prototype scope. Please note that a DAO would not typically be configured as a prototype, since a typical DAO would not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.



To define a bean as a prototype in XML, you would write configuration like so:

```
<! -- using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService"
scope="prototype"/>
```

```
<! -- the following is equivalent and preserved for backward compatibility in springbeans.
dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService"
singleton="false"/>
```

There is one quite important thing to be aware of when deploying a bean in the prototype scope, in that the lifecycle of the bean changes slightly. Spring cannot (and hence does not) manage the complete lifecycle of a prototype bean: the container instantiates, configures, decorates and otherwise assembles a prototype object, hands it to the client and then has no further knowledge of that prototype instance. This means that while initialization lifecycle callback methods will be (and are) called on all objects regardless of scope, in the case of prototypes, any configured destruction lifecycle callbacks will not be called. It is the responsibility of the client code to clean up prototype scoped objects and release any expensive resources that the prototype bean(s) are holding onto. (One possible way to get the Spring container to release resources used by singleton-scoped beans is through the use of a bean post processor which would hold a reference to the beans that need to be cleaned up.)

In some respects, you can think of the Spring container's role when talking about a prototype-scoped bean as somewhat of a replacement for the Java 'new' operator. Any lifecycle aspects past that point have to be handled by the client.

Backwards compatibility note: specifying the lifecycle scope in XML

If you are referencing the 'spring-beans.dtd' DTD in a bean definition file(s), and you are being explicit about the lifecycle scope of your bean(s) you must use the "singleton" attribute to express the lifecycle scope (remembering that the singleton lifecycle scope is the default). If you are referencing the 'spring-beans-2.0.dtd' DTD or the Spring 2.0 XSD schema, then you will need to use the "scope" attribute (because the "singleton" attribute was removed from the definition of the new DTD and XSD files in favour of the "scope" attribute).

To be totally clear about this, this means that if you use the "singleton" attribute in an XML bean definition then you must be referencing the 'spring-beans.dtd' DTD in that file. If you are using the "scope" attribute then you must be referencing either the 'spring-beans-2.0.dtd' DTD or the 'spring-beans-2.0.xsd' XSD in that file.

The other scopes

The other scopes, namely request, session, and global session are for use only in web - based applications (and can be used irrespective of which particular web application framework you are using, if indeed any). In the interest of keeping related concepts together in one place in the reference documentation, these scopes are described here.

Note The scopes that are described in the following paragraphs are only available if you are using a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you try using these next scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, you will get an IllegalStateException complaining about an unknown bean scope.

The request scope

Consider the following bean definition:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

With the above bean definition in place, the spring container will create a brand new instance of the LoginAction bean using the 'loginAction' bean definition for each and every HTTP request. That is, the 'loginAction' bean will be effectively scoped at the HTTP request level. You can change or dirty the internal state of the instance that is created as much as you want, safe in the knowledge that other requests that are also using instances created off the back of the same 'loginAction' bean definition will not be seeing these

changes in state since they are particular to an individual request. When the request is finished processing, the bean that is scoped to the request will be discarded.

The session scope

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

With the above bean definition in place, the spring container will create a brand new instance of the UserPreferences bean using the 'userPreferences' bean definition for the lifetime of a single HTTP Session. In other words, the 'userPreferences' bean will be effectively scoped at the HTTP Session level. Just like request-scoped beans, you can change the internal state of the instance that is created as much as you want, safe in the knowledge that other HTTP Session instances that are also using instances created off the back of the same 'userPreferences' bean definition will not be seeing these changes in state since they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session will also be discarded.

PropertyOverrideConfigurer

The PropertyOverrideConfigurer, another bean factory post-processor, is similar to the PropertyPlaceholderConfigurer, but in contrast to the latter, the original definitions can have default values or no values at all for bean properties. If an overriding Properties file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean factory definition is not aware of being overridden, so it is not immediately obvious when looking at the XML definition file that the override configurer is being used. In case that there are multiple

PropertyOverrideConfigurer instances that define different values for the same bean property, the last one will win (due to the overriding mechanism).

Properties file configuration lines are expected to be in the format:

```
beanName.property=value
```

An example properties file might look like this:

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

This example file would be usable against a container definition which contains a bean called dataSource, which has driver and url properties.

Note that compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors).

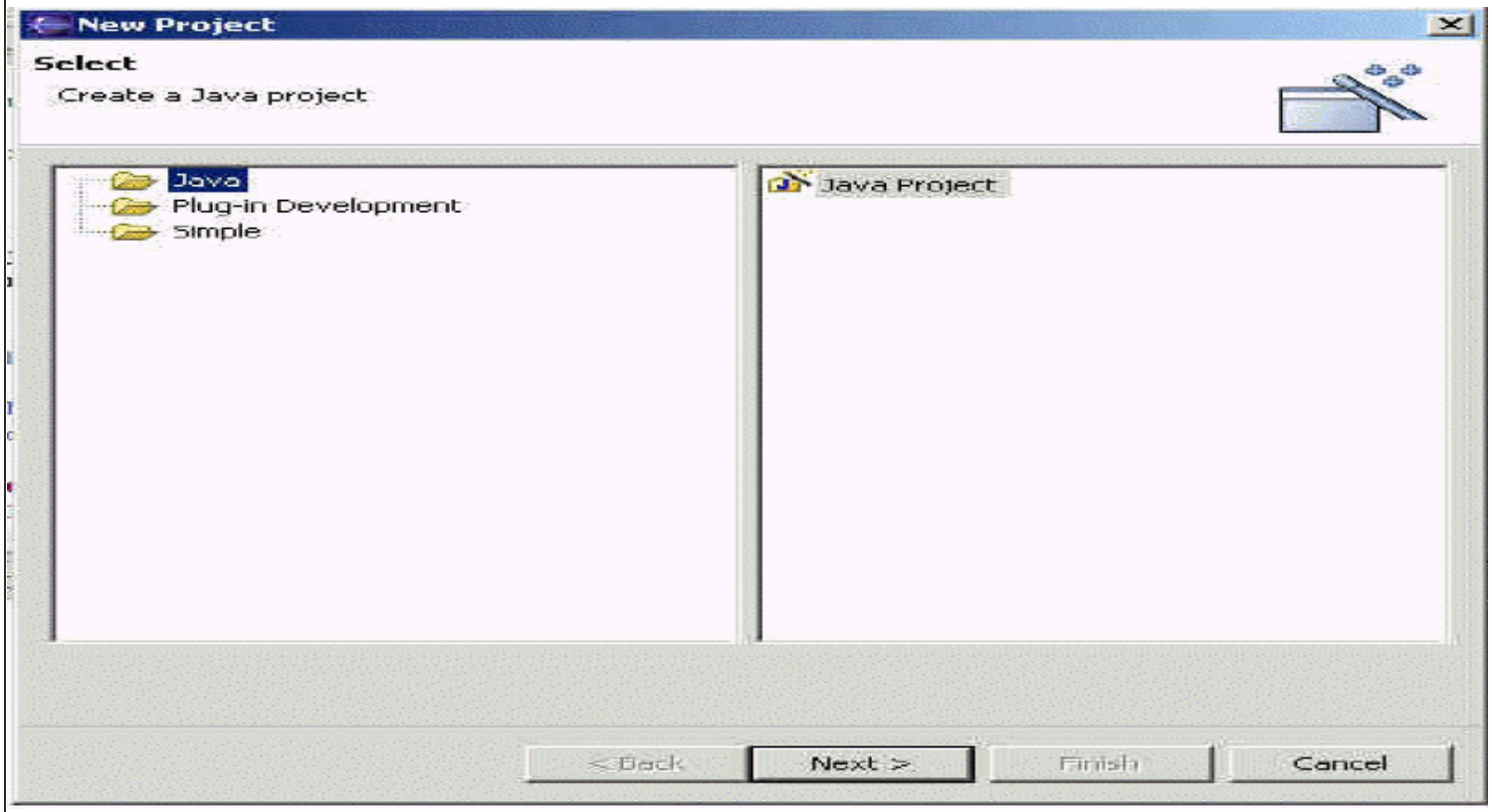
In this example...

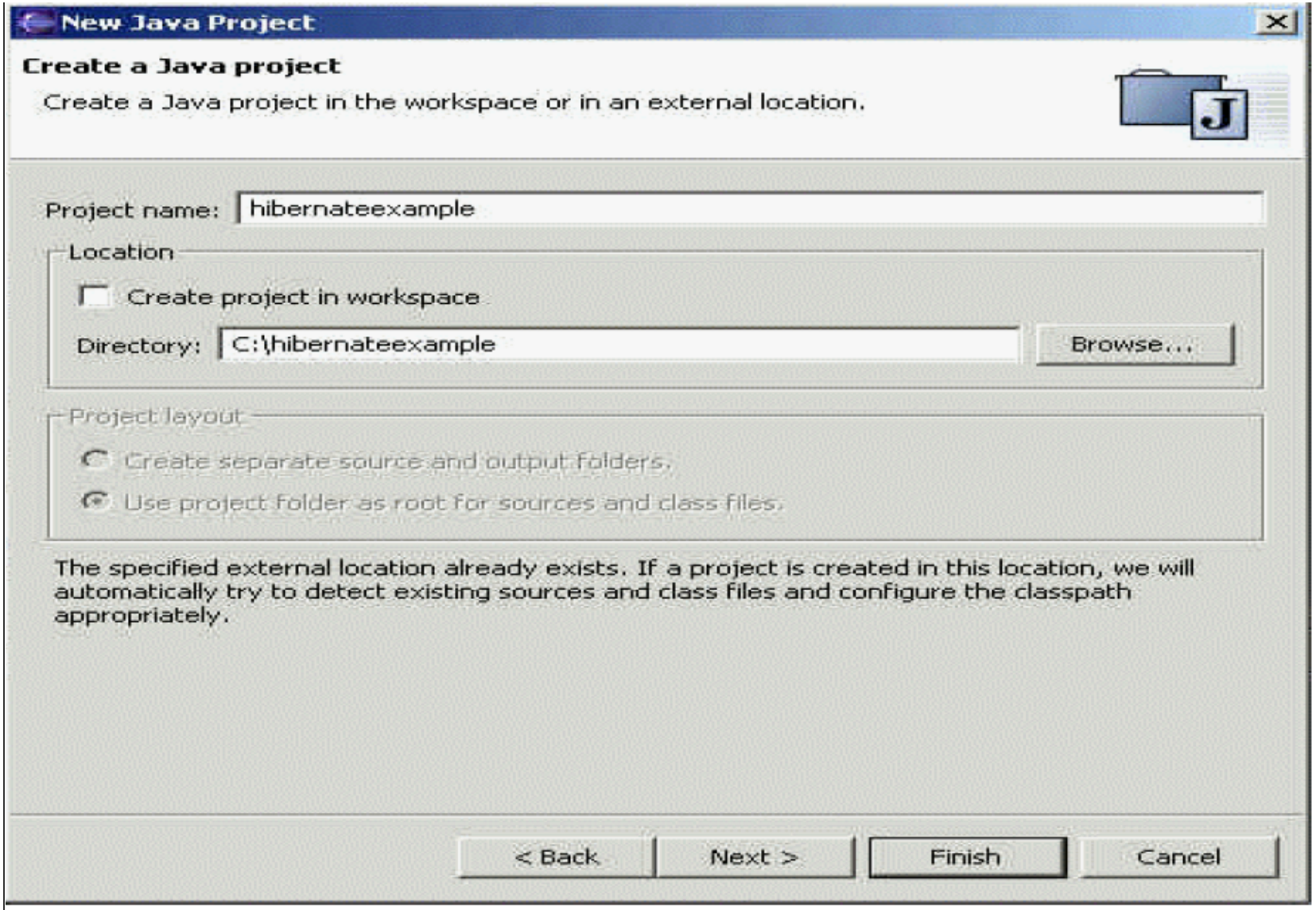
```
foo.fred.bob.sammy=123
```

... the sammy property of the bob property of the fred property of the foo bean is being set to the scalar value 123.

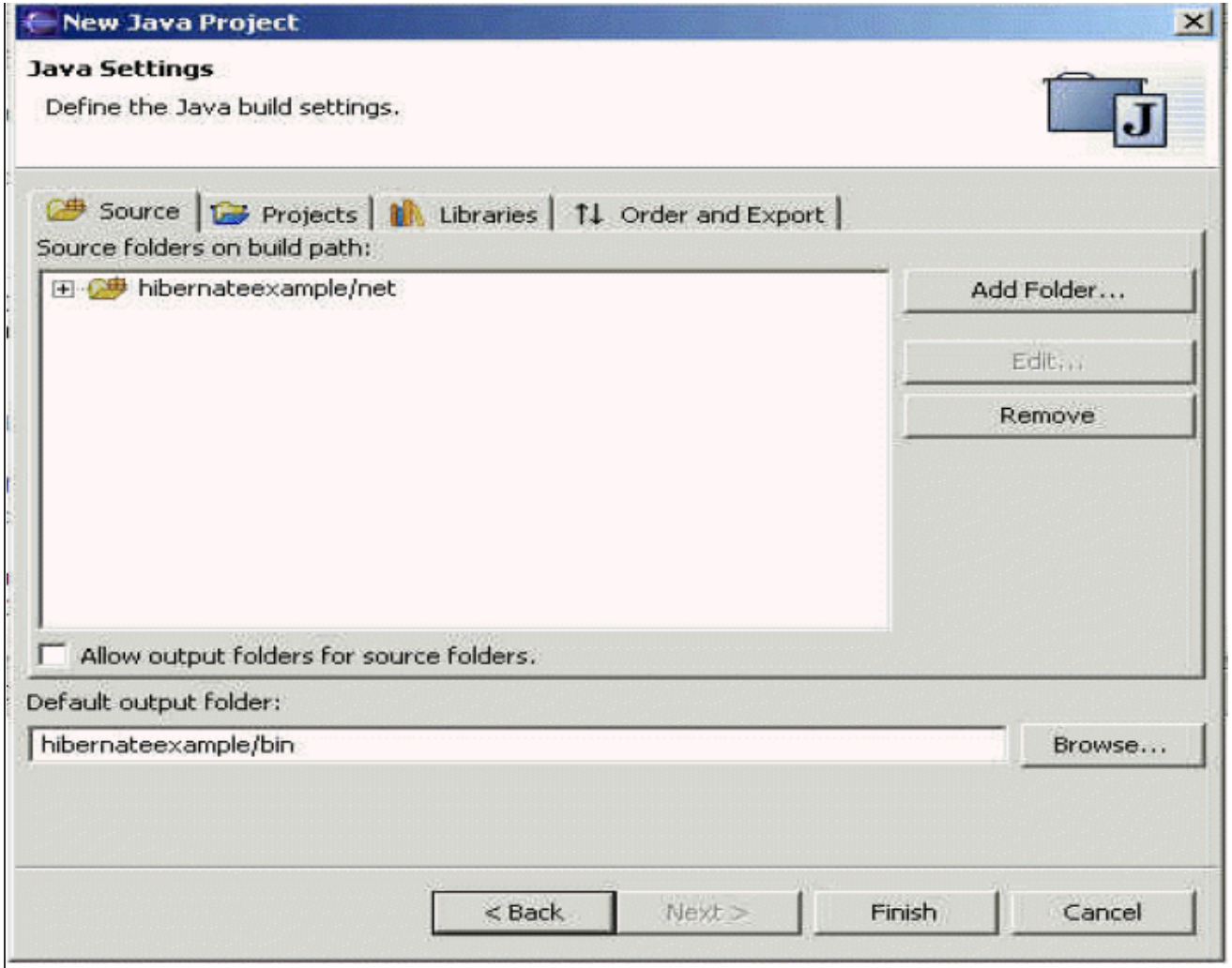
HIBERNATE

Running Hibernate Example in eclipse


New Project
Select
Create a Java project
Java
Plug-in Development
Simple
Java Project
< Back
Next >
Finish
Cancel

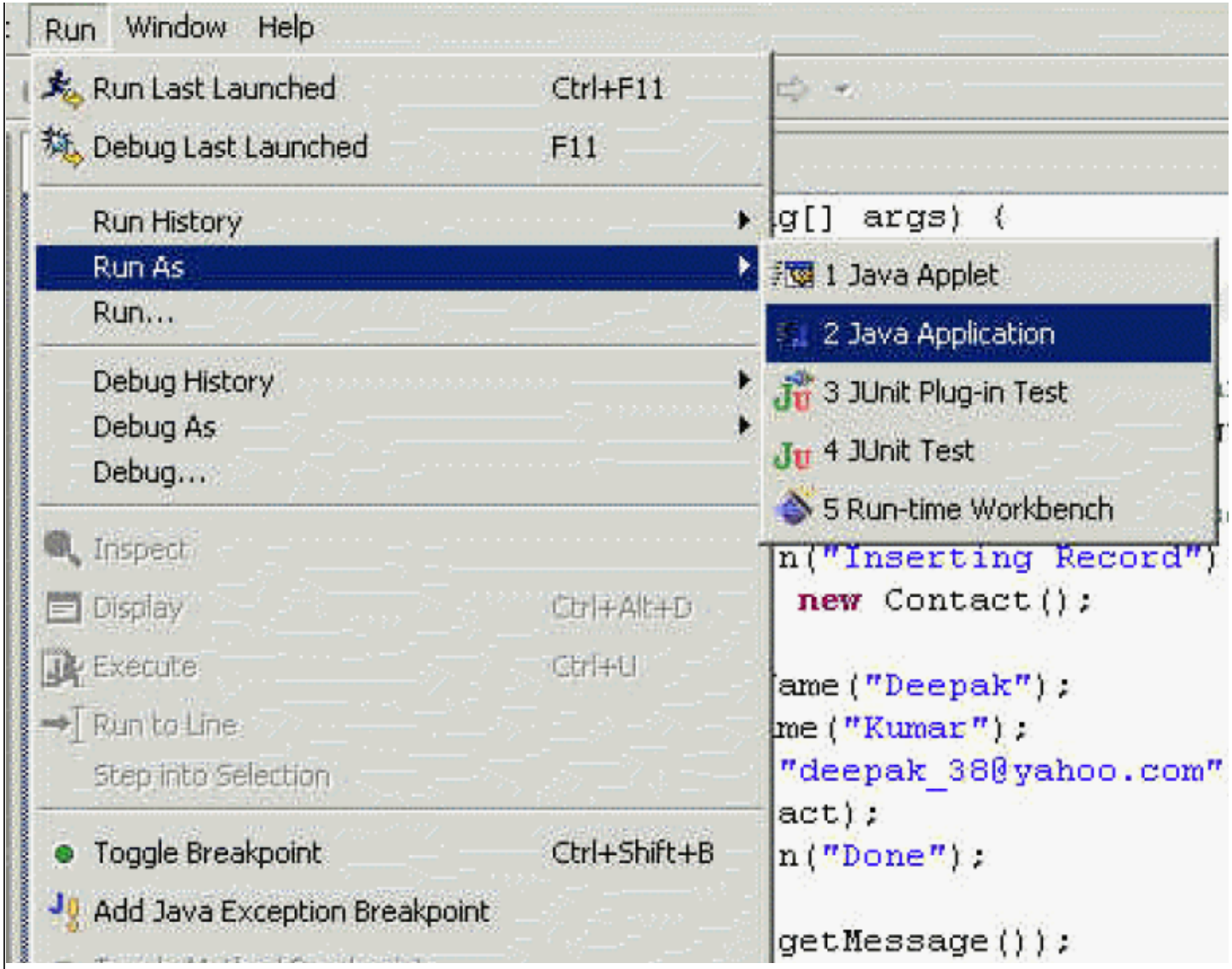


Click on "Next" button. In the next screen leave the output folder as default "hibernateexample/bin".

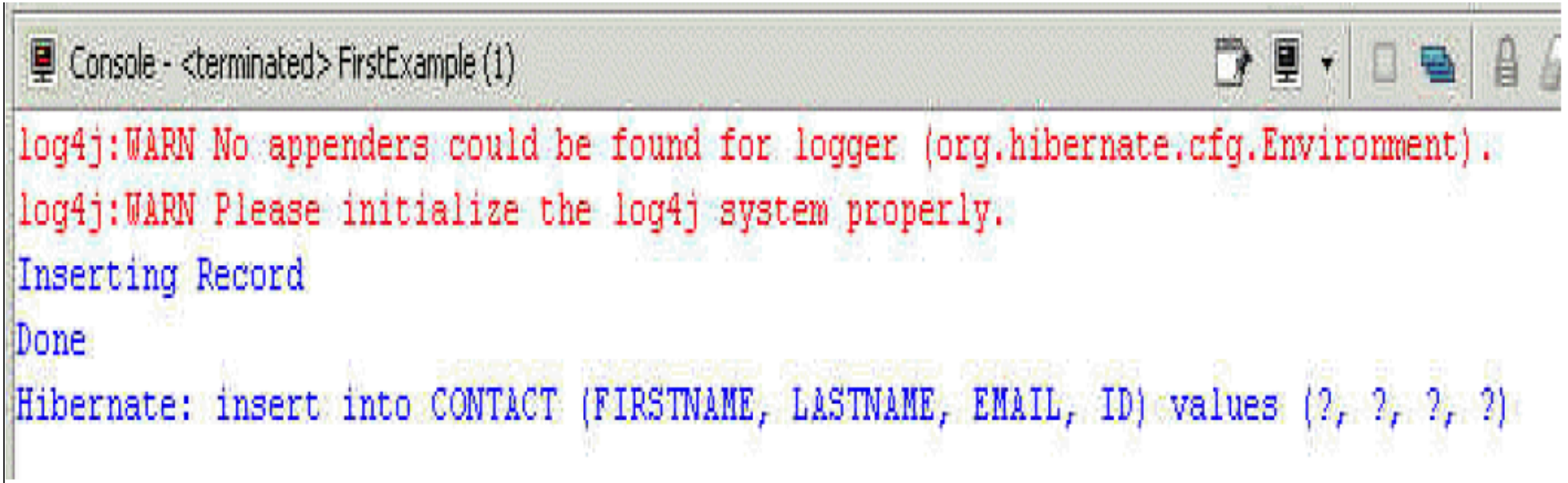


Click on the "Finish" button. Now Open the FirstExample.java in the editor as show below.

Copy contact.hbm.xml, and hibernate.cfg.xml in the bin directory of the project using windows explorer. To run the example select Run-> Run As -> Java Application from the menu bar as shown below.



This will run the Hibernate example program in Eclipse following output will displayed on the Eclipse Console.



To Delete a Record Using HQL

```
/*Table structure for table `insurance` */

drop table if exists `insurance`;

CREATE TABLE `insurance` (
`ID` int(11) NOT NULL default '0',
`insurance_name` varchar(50) default NULL,
`invested_amount` int(11) default NULL,
`investement_date` datetime default NULL,
PRIMARY KEY (`ID`)
) TYPE=MyISAM;
```

```
/*Data for the table `insurance` */

insert into `insurance` values(1,'Car Insurance',1000,'2005-01-05 00:00:00');
insert into `insurance` values(2,'Life Insurance',100,'2005-10-01 00:00:00');
insert into `insurance` values(3,'Life Insurance',500,'2005-10-15 00:00:00');
insert into `insurance` values(4,'Car Insurance',2500,'2005-01-01 00:00:00');
insert into `insurance` values(5,'Dental Insurance',500,'2004-01-01 00:00:00');
insert into `insurance` values(6,'Life Insurance',900,'2003-01-01 00:00:00');
insert into `insurance` values(7,'Travel Insurance',2000,'2005-02-02 00:00:00');
insert into `insurance` values(8,'Travel Insurance',600,'2005-03-03 00:00:00');
insert into `insurance` values(9,'Medical Insurance',700,'2005-04-04 00:00:00');
insert into `insurance` values(10,'Medical Insurance',900,'2005-03-03 00:00:00');
insert into `insurance` values(11,'Home Insurance',800,'2005-02-02 00:00:00');
insert into `insurance` values(12,'Home Insurance',750,'2004-09-09 00:00:00');
insert into `insurance` values(13,'Motorcycle Insurance',900,'2004-06-06 00:00:00');
insert into `insurance` values(14,'Motorcycle Insurance',780,'2005-03-03 00:00:00');
```

```
package durgasoft.tutorial.hibernate;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class DeleteHQLExample {
public static void main(String[] args) {
// TODO Auto-generated method stub
Session sess = null;
try {
SessionFactory fact = new Configuration().configure().buildSessionFactory();
sess = fact.openSession();
String hql = "delete from Insurance insurance where id = 1";
Query query = sess.createQuery(hql);
```

```
int row = query.executeUpate();
if (row == 0){
System.out.println("Doesn't deleted any row!");
}
else{
System.out.println("Deleted Row: " + row);
}
sess.close();
}
catch(Exception e){
System.out.println(e.getMessage());
}
}
}
```

Criteria Interface provides the following methods:

Method	Description
add	The Add method adds a Criterion to constrain the results to be retrieved.
addOrder	Add an Order to the result set.
createAlias	Join an association, assigning an alias to the joined entity
createCriteria	This method is used to create a new Criteria, "rooted" at the associated entity.
setFetchSize	This method is used to set a fetch size for the underlying JDBC query.
setFirstResult	This method is used to set the first result to be retrieved.
setMaxResults	This method is used to set a limit upon the number of objects to be retrieved.
uniqueResult	This method is used to instruct the Hibernate to fetch and return the unique records from database.

Using the Hibernate Criteria API

The Hibernate Criteria API provides an elegant way of building on-the-fly dynamic queries on Hibernate-persisted databases. Using this technique, the previous 24-line

example can be coded more consicely and more clearly using a mere 8 lines of code :

```
Criteria criteria = session.createCriteria(Sale.class);
if (startDate != null) {
criteria.add(Expression.ge("date",startDate);
}
if (endDate != null) {
criteria.add(Expression.le("date",endDate);
}
List results = criteria.list();
```

Let's have a look at the use of the Hibernate Criteria API in more detail.

Creating and using the Hibernate Criteria object

A Criteria object is created using the createCriteria() method in the Hibernate session object :

```
Criteria criteria = session.createCriteria(Sale.class);
```

Once created, you add Criterion objects (generally obtained from static methods of the Expression class) to build the query. Methods such as setFirstResult(), setMaxResults(), and setCacheable() may be used to customize the query behaviour in the same way as in the Query interface.

Finally, to execute the query, the list() (or, if appropriate uniqueResult()) method is invoqued :

```
List sales = session.createCriteria(Sale.class)
List sales = session.add(Expression.ge("date",startDate);
```

```
List sales = session.add(Expression.le("date",endDate);
List sales = session.addOrder( Order.asc("date") )
List sales = session.setFirstResult(0)
List sales = session.setMaxResults(10)
List sales = session.list();
```

Expressions

The Hibernate Criteria API supports a rich set of comparaison operators.

The standard SQL operators (=, <, ?, >, ?) are supported respectively by the following methods in the Expression class : **eq(), lt(), le(), gt(), ge()** :

```
session.createCriteria(Sale.class)
session.add(Expression.lt("date",salesDate))
session.list();

session.createCriteria(Sale.class)
session.add(Expression.eq("product",someProduct))
session.list();
```

Note that, as in HQL and unlike in a JDBC-based query, you are free to use business objects as query parameters, without having to use primary and foriegn key references.

The API also provides additional comparaison operators : like, between, in, isNull, isNotNull...

```
session.createCriteria(Sale.class)
session.add(Expression.between("date", startDate, endDate))
```

```
session.list();

session.createCriteria(Product.class)
session.add(Expression.like("A%"))
session.list();

session.createCriteria(Product.class)
session.add(Expression.in("color",selectedColors))
session.list();
```

In addition, the API provides convenient operators which : eqProperty, ltProperty, etc?

```
session.createCriteria(Sale.class)
session.eqProperty("saleDate","releaseDate")
session.list();
```

Ordering results

In HQL (and SQL), the order by clause allows you to order your query results. Using the Query API, this is done using the addOrder() method and the Order class

```
session.createCriteria(Sale.class)
session.add(Expression.between("date", startDate, endDate))
session.addOrder( Order.desc("date") )
session.addOrder( Order.asc("product.number") )
session.list();
```

Joining tables

When writing HQL queries, join clauses are often necessary to optimise the query using a "left join fetch" clause, as in the following example (I discuss this type of optimisation in another article.)

```
 from Sale sale
where sale.date > :startDate
left join fetch sale.product
```

When using the criteria API, you can do the same thing using the **setFetchMode()** function:

```
session.createCriteria(Sale.class)
session.setFetchMode("product",FetchMode.EAGER)
session.list();
```

Imagine the case of an online shop which sells shirts. Each shirt model comes in a certain number of available sizes. You want a query to find all the shirt models with sizes over 40.

In HQL, the query might be the following :

```
from Shirt shirt
join shirt.availableSizes size
where size.number > 40
```

Using the Criteria API, you use the createCriteria() to create an inner join between the two tables, as in the following example :

```
session.createCriteria(Shirt.class)
session.createCriteria("availableSizes")
session.add(Expression.gt("number", new Integer(40)))
session.list();
```

Another way of doing this is to use the createAlias() method, which does not involve creating a new instance of the Criteria class.

```
session.createCriteria(Shirt.class)
session.createAlias("availableSizes","size")
```

```
session.add(Expression.gt("size.number", new Integer(40)))
session.list();
```

Note that in both these cases, the availableSizes collection in each Shirt object will not be initialised: it is simply used as part of the search criteria.

When the Hibernate Criteria API is not appropriate

As we can see, the Hibernate Criteria API is without doubt tailor-made for dynamic query generation. It is worth noting however that there are many places where its use is not appropriate, and indeed will create code which is more complex and harder to maintain than using a standard HQL query. If the query to be executed does not involve dynamic construction (that is, the HQL query can be externalised as a named query in the Hibernate mapping files), than the use of the Hibernate Criteria API is probably not appropriate. When possible, externalising static queries presents a number of advantages:

Externalised queries can be audited and optimised if necessary by the DBA

Named queries stored in the Hibernate mapping files are easier to maintain than queries scattered through the Java code

Hibernate Named Queries are easy to cache if necessary
Criteria With the '**like**' condition
****

```
package durgasoft.tutorial.hibernate;
import org.hibernate.Session;
import org.hibernate.*;
import org.hibernate.criterion.*;
import org.hibernate.cfg.*;
import java.util.*;
```

```
public class HibernateCriteriaQueryExample2 {
public static void main(String[] args) {
Session session = null;
try {
// This step will read hibernate.cfg.xml and prepare hibernate for
// use
SessionFactory sessionFactory= new Configuration().configure().buildSessionFactory();
session = sessionFactory.openSession();
//Criteria Query Example
Criteria crit =session.createCriteria(Insurance.class);
crit.add(Restrictions.like("insuranceName", "%a%")); //Like condition
crit.setMaxResults(5); //Restricts the max rows to 5
List insurances = crit.list();
for(Iterator it =
insurances.iterator();it.hasNext();){
Insurance insurance =
(Insurance) it.next();
System.out.println(" ID: " + insurance.getLngInsuranceId());
System.out.println(" Name: " + insurance.getInsuranceName());
}
session.close();
} catch (Exception e) {
System.out.println(e.getMessage());
} finally {
}
}
}
```

Criteria With '**between**' Condition:

```
package durgasoft.tutorial.hibernate;
import org.hibernate.Session;
import org.hibernate.*;
```

```
import org.hibernate.criterion.*;
import org.hibernate.cfg.*;
import java.util.*;
public class
HibernateCriteriaQueryBetweenTwoInteger {
public static void main(String[] args) {
Session session = null;
try {
// This step will read
hibernate.cfg.xml and prepare hibernate for
// use
SessionFactory
sessionFactory = new Configuration().configure()
.buildSessionFactory();
session =
sessionFactory.openSession();
//Criteria Query Example
Criteria crit = session.createCriteria(Insurance.class);
crit.add(Expression.between("investementAmount", new Integer(
1000), new Integer(2500))); //
Between condition
crit.setMaxResults(5); //
Restricts the max rows to 5
List insurances = crit.list();
for(Iterator it =
insurances.iterator();it.hasNext();){
Insurance insurance =
(Insurance) it.next();
System.out.println("
ID: " + insurance.getLngInsuranceId());
System.out.println("
Name: " + insurance.getInsuranceName());
System.out.println("
```

```
Amount: " + insurance.getInvestementAmount());
}
session.close();
} catch (Exception e) {
System.out.println(e.getMessage());
} finally {
}
}
}
```

If U have time go through these sites for Criteria :

http://www.java2s.com/Code/Java/Hibernate/CatalogHibernate.htm

Restrictions:

1. Using criteria queries

Hibernate is providing an alternative method called "Criteria Queries" for this task. It is a set of API for us to build up a query in a more programmatic way comparing to HQL.

Sometimes we need to build up a query dynamically in our application, e.g. for an advanced search function. The traditional method of doing this is to generate a HQL statement, or SQL statement if not using Hibernate, by string concatenation. The problem for this method is making your code hard to maintain because of the hard reading statement fragments.

```
Criteria criteria = session.createCriteria(Book.class)
criteria.add(Restrictions.eq("name", "Hibernate Quickly"));
List books = criteria.list();
```

This criteria query corresponds to the following HQL query.

```
from Book book
where book.name = 'Hibernate Quickly'
```

Most methods in the Criteria class return the instance of itself, so that we can build up our criteria in the following way.

```
Criteria criteria = session.createCriteria(Book.class)
.add(Restrictions.eq("name", "Hibernate Quickly"));
List books = criteria.list();
```

2. Restrictions

We can add a series of restrictions to our criteria query to filter the results, just like building up the where clause in HQL. The Restrictions class provides a variety of methods for restriction building.

Each restriction added will be treated as a logical conjunction.

```
Criteria criteria = session.createCriteria(Book.class)
.add(Restrictions.like("name", "%Hibernate%"))
.add(Restrictions.between("price", new Integer(100), new Integer(200)));
List books = criteria.list();
```


So, this criteria query corresponds to the following HQL query.

```
from Book book where (book.name like '%Hibernate%') and (book.price between 100 and 200)
```

We can also group some of the restrictions with logical disjunction.

```
Criteria criteria = session.createCriteria(Book.class)
.add(Restrictions.or(
Restrictions.like("name", "%Hibernate%"),
Restrictions.like("name", "%Java%")
)
)
.add(Restrictions.between("price", new Integer(100), new
Integer(200)));
List books = criteria.list();
```

The corresponding HQL statement is as follow.

```
from Book book
where (book.name like '%Hibernate%' or book.name like '%Java%') and (book.price between 100
and 200)
```

3. Associations

In HQL, we can reference an association property by its name to trigger an implicit join. But for criteria queries, will the same thing happen?

```
Criteria criteria = session.createCriteria(Book.class)
.add(Restrictions.like("name", "%Hibernate%"))
.add(Restrictions.eq("publisher.name", "Manning"));
List books = criteria.list();
```

If you execute the above code, an exception will be thrown for Hibernate could not resolve the property "**publisher.name**". That means implicit join is not supported for criteria queries. To join an association explicitly, you need to create a new criteria object for it.

```
Criteria criteria = session.createCriteria(Book.class)
.add(Restrictions.like("name", "%Hibernate%"))
.createCriteria("publisher")
.add(Restrictions.eq("name", "Manning"));
List books = criteria.list();
```

```
from Book book
where book.name like '%Hibernate%' and book.publisher.name =
'Manning'
```

When using criteria query, you can specify the fetch mode of an association dynamically.

```
Criteria criteria = session.createCriteria(Book.class)
.add(Restrictions.like ("name", "%Hibernate%"))
.setFetchMode("publisher", FetchMode.JOIN)
.setFetchMode("chapters", FetchMode.SELECT);
List books = criteria.list();
```

4. Projections

If you want to customize the fields of the result, you should use Projections to specify the fields to be returned.

```
Criteria criteria = session.createCriteria(Book.class)
.setProjection(Projections.property("name"));
List books = criteria.list();
```

```
select book.name
from Book book
```

You can use the aggregate functions as well. These functions are already encapsulated in the Projections class as static methods.

```
Criteria criteria = session.createCriteria(Book.class)
.setProjection(Projections.avg("price"));
List books = criteria.list();
```

```
select avg(book.price)
from Book book
```

5. Ordering and grouping

The results can be sorted in ascending or descending order also.

```
Criteria criteria = session.createCriteria(Book.class)
.addOrder(Order.asc("name"))
.addOrder(Order.desc("publishDate"));
List books = criteria.list();
```

```
from Book book
order by book.name asc, book.publishDate desc
```

Grouping is also supported for criteria queries. You can assign any properties as group properties and they will appear in the group by clause.

```

Criteria criteria = session.createCriteria(Book.class)
.setProjection(Projections.projectionList()
.add(Projections.groupProperty("publishDate"))
.add(Projections.avg("price"))
);
List books = criteria.list();
```

```
select book.publishDate, avg(book.price)
from Book book
group by book.publishDate
```

6. Querying by example

Criteria queries can be constructed through an "example" object. The object should be an instance of the persistent class you want to query. All null properties are ignored by default.

```
Book book = new Book();
book.setName("Hibernate");
book.setPrice(new Integer(100));
Example exampleBook = Example.create(book);
Criteria criteria = session.createCriteria(Book.class)
.add(exampleBook);
List books = criteria.list ();
```

```
from Book book
where book.name = 'Hibernate' and book.price = 100
```

You can specify some creation rules for the construction of example object. For example, exclude some of the properties or enable "like" for string comparison.

```
Book book = new Book();
book.setName("%Hibernate%");
book.setPrice(new Integer(100));
Example exampleBook = Example.create(book)
.excludeProperty("price")
.enableLike();
Criteria criteria = session.createCriteria(Book.class)
.add(exampleBook);
List books = criteria.list();
```

```
from Book book where book.name like '%Hibernate%'
```

Criteria Code:

```
List cats = session.createCriteria(Cat.class)
.add( Restrictions.like("name", "Iz%") )
.add( Restrictions.gt( "weight", new Float(minWeight) ) )
.addOrder(Order.asc("age") )
.list();
```

You may navigate associations using **createAlias()** or **createCriteria()**.

```
List cats = session.createCriteria(Cat.class)
.createCriteria("kittens")
.add( Restrictions.like("name", "Iz%") )
.list();
LiNag Picsst cats = session.createCriteria(Cat.class)
.createAlias("kittens", "kit")
.add( Restrictions.like("kit.name", "Iz%") )
.list();
```

You may specify projection and aggregation using Projection instances obtained via the factory methods on Projections.

```
List cats = session.createCriteria(Cat.class)
.setProjection( Projections.projectionList()
.add( Projections.rowCount() )
.add( Projections.avg("weight") )
.add( Projections.max("weight") )
.add( Projections.min("weight") )
.add( Projections.groupProperty("color") )
)
```

```
.addOrder( Order.asc("color") )
.list();
```

Hibernate API

This documentation concentrates upon the following Core API interfaces:

- org.hibernate.Hibernate
- org.hibernate.Session
- org.hibernate.SessionFactory
- org.hibernate.Transaction
- org.hibernate.Query
- org.hibernate.Criteria
- org.hibernate.cfg.Configuration
- org.hibernate.expression.Expression
- org.hibernate.expression.Order
- org.hibernate.criterion.Restrictions

The API of Restrictions Class is in the url :
http://www.hibernate.org/hib_docs/v3/api/org/hibernate/criterion/Restrictions.html

Relations :

Many-to-Many Mapping:

Hibernate.cfg.xml:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:servicename
</property>
<property name="connection.username">scott</property>
<property name="connection.password">tiger</property>
<mapping resource="Order.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

Order.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Order" table="orders" >
<id name="oid" >
<generator class="assigned" />
</id>
<property name="cname" />
<property name="odate" />
<set name="products" table="order_product" cascade="all" >
<key column="order_id" />
<many-to-many class="Product" column="product_id" />
</set>
</class>
```

```
<class name="Product" table="products" >
<id name="pid" >
<generator class="assigned" />
</id>
<property name="pname" />
<property name="price" />
<set name="orders" table="order_product" cascade="all">
<key column="product_id" />
<many-to-many class="Order" column="order_id" />
</set>
</class>
</hibernate-mapping>
```

Product.java

```
import java.util.*;
public class Product
{ private int pid;
private String pname;
private double price;
private Set orders=new HashSet();
public void setPid(int n){ pid=n; }
public int getPid(){ return pid; }
public void setPname(String s){ pname=s; }
public String getPname(){ return pname; }
public void setPrice(double d){ price=d; }
public double getPrice(){ return price; }
public void setOrders(Set s){
// System.out.println("setorders called");
orders=s;
if(s==null)
{
break;
```

```
}
/*
Iterator i=s.iterator();
while(i.hasNext())

{
Order o=(Order)i.next();
System.out.println(" value :"+o.getCname()+" value");
}
System.out.println("****************************************");
*/
}
public Set getOrders(){
if(orders==null)
{
}
System.out.println("getorders called");
return orders; }
}
```

Order.java

```
import java.util.*;
public class Order
{
private int oid;
private String cname;
private String odate;
private Set products=new HashSet();
public void setOid(int n){ oid=n; }
public int getOid(){ return oid; }
public void setCname(String s){ cname=s; }
public String getCname(){ return cname; }
```

```
public void setOdate(String d){ odate=d; }
public String getOdate(){ return odate; }
public void setProducts(Set s){ products=s; }
public Set getProducts(){ return products; }
}
```

```
/*
create table Orders
(oid number(5) primary key,
cname varchar2(20),
odate varchar2(12)
);
create table products
(pid number(5) primary key,
pname varchar2(20),
price number(8,2)
);

create table order_product
( order_id number(5) references orders(oid),
product_id number(5) references products(pid),
primary key(order_id,product_id)
);
*/
```

Saveorder.java :

```
import java.io.*;
import java.util.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class saveorder
```

```
{
public static void main(String[] args)
{
SessionFactory sf=new Configuration().configure().buildSessionFactory();
Session ses=sf.openSession();
Transaction t=ses.beginTransaction();
Product p1=new Product();
p1.setPid(3000);
p1.setPname("PRODUCT1");
p1.setPrice(5050);
Product p2=new Product();
p2.setPid(3001);
p2.setPname("PRODUCT2");
p2.setPrice(6060);
Product p3=new Product();
p3.setPid(3002);
p3.setPname("PRODUCT3");
p3.setPrice(7070);
Set s=new HashSet();
s.add(p1);
s.add(p2);
s.add(p3);
Order o1=new Order();
o1.setOid(1001);
o1.setCname("ali");
o1.setOdate("march");
o1.setProducts(s);
ses.save(o1);
t.commit();
ses.close();
}
}
```

Attribute 'cascade' : specifies which operations should be cascaded from the parent object to the associated object.

One to One Mapping Example :

Hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<!-- Database connection settings -->
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property
name="connection.url">jdbc:oracle:thin:@localhost:1521:ORACLE</property>
<property name="connection.username">scott</property>
<property name="connection.password">tiger</property>
<mapping resource="bank.hbm.xml"/>
<mapping resource="address.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Address.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Address" table="address_tab">
```

```
<id name="accno" type="int">
<column name="accountno"/>
<generator class="increment"/>
</id>
<property name="street" column="street"/>
<property name="city" column="city"/>
<property name="state" column="state"/>
<property name="pin" column="pin" type="int"/>
</class>
</hibernate-mapping>
```

Bank.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="bank" table="bank_tab">
<id name="accno" type="int">
<column name="accountno"/>
<generator class="increment"/>
</id>
<one-to-one name="address" class="Address" cascade="all"/>
<property name="name" column="name"/>
<property name="acctype" column="acctype"/>
</class>
</hibernate-mapping>
```

Address.java

```
public class Address
{
```

```
String street,city,state;
int pin;
int accno;
public void setAccno(int accno)
{
this.accno=accno;
}
public int getAccno()
{
return accno;
}
public void setStreet(String street)
{
this.street=street;
}
public String getStreet()
{
return street;
}
public void setCity(String city)
{
this.city=city;
}
public String getCity()
{
return city;
}
public void setState(String state)
{
this.state=state;
}
public String getState()
{
```

```
return state;
}
public void setPin(int pin)
{
this.pin=pin;
}
public int getPin()
{
return pin;
}
}
```

Bank.java

```
public class bank
{
int accno;
String name;
Address ad;
String acctype;
public void setAccno(int i)
{
accno=i;
}
public int getAccno()
{
return accno;
}
public void setName(String s)
{
name=s;
}
public String getName()
```

```
{
return name;
}
public void setAcctype(String i)
{
acctype=i;
}
public String getAcctype()
{
return acctype;
}
public void setAddress(Address ad)
{
this.ad=ad;
}
public Address getAddress()
{
return ad;
}
}
```

getAccountDetails.java

```
import org.hibernate.*;
import org.hibernate.cfg.*;
public class getAccountDetails
{
public static void main(String[] args)
{
Configuration cfg=new Configuration().configure();
SessionFactory sf=cfg.buildSessionFactory();
Session ses=sf.openSession();
bank b=new bank();
```

```
Integer accno=new Integer(args[0]);
ses.load(b,accno);
System.out.println("name :"+b.getName());
System.out.println("acctype :"+b.getAcctype());
Address ad=b.getAddress();
System.out.println(ad.getStreet());
System.out.println(ad.getCity());
System.out.println(ad.getState());
System.out.println(ad.getPin());
ses.close();}}
```

```
<one-to-one name="shippingAddress"
class="Address"
cascade="all"/>
```

Many to One Mapping Example:

Hibernate.cfg.xml:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<!-- Database connection settings -->
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:ORACLE</property>
<property name="connection.username">scott</property>
<property name="connection.password">tiger</property>
<mapping resource="bank.hbm.xml"/>
<mapping resource="address.hbm.xml"/>
```

```
</session-factory>
</hibernate-configuration>
```

Bank.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="bank" table="bank_tab">
<id name="accno" type="int">
<column name="accountno"/>
</id>
<many-to-one name="hai" class="Address" column="acctype" cascade="all" />
<property name="name" column="name"/>
<property name="acctype" column="acctype"/>
</class>
</hibernate-mapping>
```

Address.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Address" table="address_tab1">
<id name="accno" type="int">
<column name="accountno"/>
<generator class="increment"/>
</id>
<set name="accounts">
```

```
<key column="accountno"/>
<one-to-many class="bank"/>
</set>
<property name="street" column="street"/>
<property name="city" column="city"/>
<property name="state" column="state"/>
<property name="acctype" column="acctype" type="int"/>
</class>
</hibernate-mapping>
```

Address.java:

```
import java.util.*;
public class Address
{
String street,city,state;
int acctype;
int accno;
Set accounts;
public void setAccno(int accno)
{
this.accno=accno;
}
public int getAccno()
{
return accno;
}
public void setStreet(String street)
{
this.street=street;
}
public String getStreet()
{
```

```
return street;
}
public void setCity(String city)
{
this.city=city;
}
public String getCity()
{
return city;
}
public void setState(String state)
{
this.state=state;
}
public String getState()
{
return state;
}
public void setAcctype(int acctype)
{
this.acctype=acctype;
}
public int getAcctype()
{
return acctype;
}}
```

Bank.java:

```
public class bank
{
int accno,acctype;
String name;
```

```
int hai;
public void setAccno(int i)
{
accno=i;
}
public int getAccno()
{
return accno;
}
public void setName(String s)
{
name=s;
}
public String getName()
{
return name;
}
public void setAcctype(int i)
{
acctype=i;
}
public int getAcctype()
{
return acctype;
}
public void sayHai(int i)
{
hai=i;
}
public int getHai()
{
return hai;
}}
```

getBankDetails.java

```
import org.hibernate.*;
import org.hibernate.cfg.*;
public class getBankDetails
{
public static void main(String[] args)
{
Configuration cfg=new Configuration().configure();
SessionFactory sf=cfg.buildSessionFactory();
Session ses=sf.openSession();
bank b=new bank();
Integer accno=new Integer(args[0]);
ses.load(b,accno);
ses.close();
System.out.println("name :"+b.getName());
System.out.println("acctype :"+b.getAcctype());
System.out.println("hai :"+b.getHai());
}}
```

Using Map in the hibernate mapping file: Hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.OracleDriver</property>
<property
name="connection.url">jdbc:oracle:thin:@localhost:1521:servicename</property>
<property name="connection.username">newuser</property>
<property name="connection.password">newuser</property>
```

```
<property name="connection.pool_size">4</property>
<property
name="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</property>
<mapping resource="Speaker.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

Speaker.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="Speaker" table="speaker1" >
<id name="sid" />
<property name="sname" />
<map name="sessions" >
<key column="speaker_id" />
<index column="sessions_index" type="string"/>
<one-to-many class="Sessions" />
</map>
<map name="phonenumbers" table="speaker_phonenumbers1" >
<key column="speaker_id" />
<index column="phone_index" type="string" />
<element type="string" column="phone" />
</map>
</class>
<class name="Sessions" table="sessions1" >
<id name="sesid" />
<property name="sesname" />
</class>
</hibernate-mapping>
```

Sessions.java

```
public class Sessions
{
private int sesid;
private String sesname;
public void setSesid(int n)
{ sesid=n; }
public int getSesid()
{
return sesid;
}

public void setSesname(String s){ sesname=s;}
public String getSesname(){ return sesname; }
}
```

Insertservlet.java

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;
public class insertservlet extends HttpServlet
{
public void service(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter pw=res.getWriter();
try{
```

```
SessionFactory sf=new Configuration().configure().buildSessionFactory();
Session ses=sf.openSession();
Transaction tx=ses.beginTransaction();
// new speakers
Speaker s1=new Speaker();
s1.setSid(1001); s1.setSname("James");
Speaker s2=new Speaker();
s2.setSid(1002); s2.setSname("Mary");
ses.save(s1);ses.save(s2);
// new sessions
Sessions t1=new Sessions();
t1.setSesid(1);t1.setSesname("session1");
Sessions t2=new Sessions();
t2.setSesid(2);t2.setSesname("session2");
Sessions t3=new Sessions();
t3.setSesid(3);t3.setSesname("session3");
ses.save(t1);ses.save(t2);ses.save(t3);
// mapping sessions to speakers
s1.getSessions().put("ses1",t1);
s1.getSessions().put("ses2",t2);
s2.getSessions().put("ses3",t3);
s1.getPhonenumbers().put("ph1","1111111");
s1.getPhonenumbers().put("ph2","222222");
s2.getPhonenumbers().put("ph3","12345678");
s2.getPhonenumbers().put("ph4","33333333");
ses.update(s1);
ses.update(s2);
tx.commit();
ses.close();
pw.println("inserted successfully...");
}catch(Exception e)
{
pw.println(e);
```

```
e.printStackTrace();
}
}
}
```

Selectservlet.java

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;
public class selectservlet extends HttpServlet
{
public void service(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter pw=res.getWriter();
Configuration con=new Configuration();
con.configure();
SessionFactory sf=con.buildSessionFactory();
Session ses=sf.openSession();
Query query=ses.createQuery("from Speaker");
Iterator i1=query.iterate();
while(i1.hasNext())
{
Speaker s1=(Speaker)i1.next();
pw.println(s1.getSid()+" "+s1.getSname()+"
");
Map sessions=s1.getSessions();
Collection cl=sessions.values();
```

```
Iterator i2=cl.iterator();
while(i2.hasNext())
{
Sessions t1=(Sessions)i2.next();
pw.println("---> "+t1.getSesid()+" "+t1.getSesname()+"
");
}
Map phones=s1.getPhonenumbers();
Iterator i3=phones.values().iterator();
while(i3.hasNext())
{
String ph=(String)i3.next();
pw.println("----> "+ph+"
");
}
pw.println("
");
}
ses.close();
}
}
```

```
/*
Map
/
public void put(key,value);
public Collection values();
\
public Iterator iterator();*/
```

```
<set name="order" order-by="ORDER_FK" lazy="false" inverse="true"
cascade="all-delete-orphan">
<key>
<column name="ORDER_FK"/>
</key>
<one-to-many class="foo.OrderImpl"/>
</set>
```

attribute : inverse

NHibernate and Inverse=True|False Attribute

In order to truly understand the inverse attribute in NHibernate, you need to first look at your relationship from the database table point of view and how your domain objects maps into their respective tables. It comes down to determining who is the owner of the relationship association between the parent and child objects.

Suppose you have ParentTable (parentId PK, parentcol1, parentcol2) and ChildTable(childid PK, parentid FK, childcol1, childcol2). In a one-to-many relationship, you will have one parent and many children. The ChildTable has a primary key and foreign key. The foreign key is a reference to the primary key in the ParentTable. If you look at the ChildTable, you will notice that the ChildTable stores extra information about a row in the ParentTable via the ChildTable's parentId foreign key. Using this perspective, the "child" owns (knows about) the relationship it has with its parent row. So, the child will be the owner of the relationship in this one-to-many relationship.

Where does inverse come in? Well, from the Parent's perspective, since the parent doesn't own the relationship it is considered the "inverse" of the relationship. Below is some additional information I pulled down an edited from IBM's websphere pages:

http://www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html

Many-to-one relationship

The entity declaring the many-to-one relationship is the child object (or owner of the relationship), as its table has the foreign key, while the object that is referenced by the entity declaring the many-to-one relationship is the parent object. Since its table doesn't have the foreign key, it is the non-owner, or inverse of the relationship.

Hibernate conventions

In Hibernate, many-to-one relationship maps as follows:

Use many-to-one element in the child class.
Define the primary key in the parent class.

One-to-many relationship

A one-to-many relationship defines a reference to a collection of objects. It is the most common kind of relationship that you will find in object models due to the fact that use cases typically require traversal from the parent object to its children, but may or may not require traversal from the child back to its parent; which means a unidirectional one-to-many relationship will suffice in most cases.

The entity declaring the one-to-many relationship is the parent object (and is the nonowner). The table for this entity defines the primary key, but it does not have the foreign key -- that is in the child.

In Hibernate, the mapping of one-to-many relationships is generally done by adding a column to the child table for the foreign key, but the details of the mapping differs depending on whether it is a unidirectional or a bidirectional one-to-many relationship.

In the unidirectional case, the foreign key column in the child table doesn't map to a property in the child object; it is in the data model, but not the object model. Since it is unidirectional there is just a property in the parent object [that contains a collection of children]; ~~not the child~~ [the child object does not contain any information back to the parent]. In addition, the foreign key column [in the child table] has to be defined as nullable because Hibernate will first insert the child row (with a NULL foreign key) and then update it [the inserted row] later [with the parent's primary key].

In the bidirectional case, the object-relational mapping is better because there is a property in the child object for the foreign key column, and that column doesn't have to be nullable in the database. But the resulting object model has cyclic dependencies and tighter coupling between the objects, and requires additional programming to set both sides of the relationship.

As you can see, there are several tradeoffs to consider with regard to the definition of one-to-many relationships, but using unidirectional relationships is generally recommended unless there are use cases that indicate the need for navigation in both directions.

Hibernate conventions

In Hibernate, one-to-many (unidirectional) relationships map as follows:

Use the set, bag, or list with one-to-many sub-element in parent class.

Create a foreign key in the table representing the child class if the relationship is unidirectional; otherwise, use the many-to-one relationship for a bidirectional relationship.

There are some additional features that are often used in the definition of one-to-many relationships[...]:
Hibernate
inverse="true"

In Hibernate, you might encounter the inverse="true" attribute being used in the definition of a bidirectional relationship. If so, don't worry, because this feature is equivalent to ... the non-owner of the relationship whose table doesn't have the foreign key. Similarly, the Hibernate inverse="false" attribute is equivalent to the ... owner of the relationship whose table has the foreign key.

In general, these notions align, except for the case where someone defines a Hibernate mapping with inverse="true" set on the many-to-one side of a bidirectional relationship. If you find such a mapping, you should change it ..., as it is not a best practice in Hibernate and does not generate optimal SQL. For instance, if the many-to-one side is set to inverse="true" then every time you create a child, Hibernate will execute two SQL statements, one to create the child and one to update the child with the foreign key of the parent. Changing it to inverse="false" on the many-to-one side and setting inverse="true" on the one-tomany side will fix that oversight ...

Many-to-many relationship

A many-to-many relationship defines a reference to a collection of objects through a mapping table. Many-to-many relationships are not all that common in an object model, but they will generally be bidirectional.

Like the other bidirectional relationships, there is an owning and a non-owning side. In this case, the owning side has the mapping table, instead of the foreign key. Either side can be designated as the owning side; it doesn't matter which side you pick. Hibernate conventions

In Hibernate, many-to-many relationship maps as follows:

The non-owner uses the collections (set, bag, or list) element with the inverse="true" attribute, the table attribute, the key sub-element, and the many-tomany sub-element.

The owner of the relationship uses the collections (set, bag, or list) element with the table attribute, the key sub-element, and the many-to-many sub-element.

1. One-to-many association

In the previous example, we treat each chapter of a book as a string and store it in a collection. Now we extend this example by making each chapter of a persistent object type. For one book object can relate to many chapter objects, we call the association from book to chapter a "one-to-many" association. We will first define this association as "unidirectional", i.e. navigable from book to chapter only, and then extend it to be "bi-directional". Remember that we have a Chapter class in our application that hasn't mapped to the database. We first create a Hibernate mapping for it and then define an auto-generated identifier. This generated identifier is efficient for associating objects.

```
public class Chapter {
private Long id;
private int index;
private String title;
private int numOfPages;
// Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Chapter" table="CHAPTER">
<id name="id" type="long" column="ID">
<generator class="native" />
</id>
<property name="index" type="int" column="IDX" not-null="true" />
<property name="title" type="string">
<column name="TITLE" length="100" not-null="true" />
</property>
<property name="numOfPages" type="int" column="NUM_OF_PAGES" />
```

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

For we have added a new persistent object to our application, we need to specify it in the Hibernate configuration file also.

```
<mapping resource="com/metaarchit/bookshop/Chapter.hbm.xml" />
```

For our Book class, we already have a collection for storing chapters, although only the titles are being stored. We can still make use of this collection but we put chapter objects instead. Which collection type should we use? As there cannot be any duplicated chapters inside a book, we choose the <set> collection type.

```
public class Book {
private Long id;
private String isbn;
private String name;
private Publisher publisher;
private Date publishDate;
private Integer price;
private Set chapters;
// Getters and Setters
}
```

To tell Hibernate that we are storing chapter objects but not strings inside the collection, we can simply use <one-to-many> instead of <element>.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
<id name="id" type="long" column="ID">
<generator class="native"/>
</id>
<property name="isbn" type="string">
```

```
<column name="ISBN" length="50" />
</property>
<property name="name" type="string">
<column name="BOOK_NAME" length="100" not-null="true" unique="true" />
</property>
<property name="publishDate" type="date" column="PUBLISH_DATE" />
<property name="price" type="int" column="PRICE" />
<set name="chapters" table="BOOK_CHAPTER">
<key column="BOOK_ID" />
<element column="CHAPTER" type="string" length="100" />
<one-to-many class="Chapter" />
</set>
</class>
</hibernate-mapping>
```

Since the chapters should be accessed sequentially, it is more sensible to sort it by the "index" property or "IDX" column. The simplest and most efficient way is to ask the database to sort for us. Page 3 of 9

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
...
<set name="chapters" order-by="IDX">
<key column="BOOK_ID" />
<one-to-many class="Chapter" />
</set>
</class>
</hibernate-mapping>
```

If we want our collection can be accessed randomly, e.g. get the tenth chapter, we can use the <list> collection type. Hibernate will use the IDX column of CHAPTER table as the list index.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
```

```
...
<list name="chapters">
<key column="BOOK_ID" />
<list-index column="IDX"/>
<one-to-many class="Chapter" />
</list>
</class>
</hibernate-mapping>
```

1.1. Lazy initialization and fetching strategies

We can also specify the lazy and fetch attributes for the association, just like we do for the collection of values previously.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
...
<set name="chapters" lazy="false" fetch="join">
<key column="BOOK_ID" />
<one-to-many class="Chapter" />
</set>
</class>
</hibernate-mapping>
```

In HQL, we can also use "left join fetch" to specify the fetching strategy and force the collection to be initialized, if it is lazy. This is the efficient way to initialize the lazy associations of all the objects returned from a query.

```
Session session = factory.openSession();
try {
Page 4 of 9
Query query = session.createQuery(
```

```
"from Book book left join fetch book.chapters where book.isbn = ?");
query.setString(0, isbn);
Book book = (Book) query.uniqueResult();
return book;
} finally {
session.close();
}
```

1.2. Cascading the association

We have not discussed collection cascading before, for there is nothing can be cascaded for a collection of values. For a <one-to-many> association, or a collection of persistent objects, we can cascade the operation to the objects inside the collection.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
...
<set name="chapters" order-by="IDX" cascade="save-update,delete">
<key column="BOOK_ID" />
<one-to-many class="Chapter" />
</set>
</class>
</hibernate-mapping>
```

You can use a single saveOrUpdate() call to persist the whole object graph now. Suppose we are persisting a book with two chapters. If you inspect the SQL statements generated by Hibernate, you may feel a little bit confusing.

```
insert into BOOK (ISBN, BOOK_NAME, PUBLISH_DATE, PRICE, PUBLISHER_ID, ID) values
(?, ?, ?, ?, ?,null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, ID) values (?, ?, ?, null)
```

```
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, ID) values (?, ?, ?, null)
update CHAPTER set BOOK_ID=? where ID=?
update CHAPTER set BOOK_ID=? where ID=?
```

The result is that three INSERT statements and two UPDATE statements have been executed in total. Why not only three INSERT statements to be executed as our expectation?
When we call saveOrUpdate() and pass in the book object graph, Hibernate will perform the following actions:

- Save or update the single book object. In our case, it should be saved because it is newly created and the ID is null.
- Cascade the saveOrUpdate() operation to each chapter in the collection. In our case, each of them will be saved for their IDs are also null.
- Persist the one-to-many association. Each row of the CHAPTER table will be given the BOOK_ID.

The two UPDATE statements seem to be unnecessary since it should be able to include the BOOK_ID in the INSERT statements.
We can solve this problem by making the association

Now let's consider another case, suppose we want to remove the third chapter from a book, we can use the following code fragment. We iterate over the chapter collection, find the third chapter and remove it. The result of this code is the BOOK_ID column of the third chapter has been set to null.

That is, it doesn't belong to a book any more.

```
for (Iterator iter = book.getChapters().iterator(); iter.hasNext();) {
Chapter chapter = (Chapter) iter.next();
if (chapter.getIndex() == 3) {
iter.remove();
```

```
}
}
```

Does this behavior make sense? The chapter object has become meaningless after removing from a book. So we should delete this object explicitly after removing it. But how about we forget to do it?

It may become garbage in our database and waste our memory. The chapter object once removed from a book is an "orphan". Hibernate is providing one more cascading type for collection of persistent objects. An object can be deleted automatically once it becomes orphan.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
...
<set name="chapters" order-by="IDX" cascade="save-update,delete,delete-orphan">
<key column="BOOK_ID" />
<one-to-many class="Chapter" />
</set>
</class>
</hibernate-mapping>
```

2. Bi-directional one-to-many / many-to-one association

In some cases, we want our associations to be bi-directional. Suppose we have a page for displaying the detail of a chapter inside a book. So we need to know which book this chapter belongs to, given a chapter object. We can do it by adding a reference to book in the Chapter class. This association is a one-to-many association. So, the book-to-chapter and chapter-to-book associations combine a bi-directional association.

```
public class Chapter {
private Long id;
private int index;
```

```
private String title;
private int numOfPages;
private Book book;
// Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Chapter" table="CHAPTER">
...
<many-to-one name="book" class="Book" column="BOOK_ID" />
</class>
</hibernate-mapping>
```

Now if we persist a book with two chapters again and inspect the SQL statements, we will find the following results.

```
insert into BOOK (ISBN, BOOK_NAME, PUBLISH_DATE, PRICE,
PUBLISHER_ID, ID) values (?, ?, ?, ?, ?,null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, BOOK_ID, ID) values
(?, ?, ?, ?, null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, BOOK_ID, ID) values
(?, ?, ?, ?, null)
update CHAPTER set BOOK_ID=? where ID=?
update CHAPTER set BOOK_ID=? where ID=?
```

Note that there are still five statements in total but the BOOK_ID is included in the INSERT statement. So the last two UPDATE statements can be omitted. We can do that by adding an "inverse" attribute to the collection. Hibernate will not persist the collection marked with inverse. But the operations will still be cascading to the persistent objects in the collection.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
```

```
...
<set name="chapters" order-by="IDX" cascade="saveupdate,
delete,delete-orphan"
inverse="true">
<key column="BOOK_ID" />
<one-to-many class="Chapter" />
</set>
</class>
</hibernate-mapping>
```

3. Many-to-many association

The last type of association we go through is the "many-to-many" association. Remember that we have used customer and address as an example when introducing "one-to-one" association. Now we extend this example to accept a "many-to-many" association. For some customers, they may have more than one address, such as home address, office address, mailing address, etc. For the staff working in the same company, they should share one office address. That is how the "many-to-many" association comes in.

```
public class Customer {
private Long id;
private String countryCode;
private String idCardNo;
private String firstName;
private String lastName;
private String email;
private Set addresses;
// Getters and Setters
}
```

Defining a <many-to-many> association is much like the <one-to-many>. For a <many-to-many> association, we must use a join table for storing the keys of both parties.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Customer" table="CUSTOMER">
<id name="id" type="long" column="ID">
<generator class="native"/>
</id>
<properties name="customerKey" unique="true">
<property name="countryCode" type="string" column="COUNTRY_CODE"
not-null="true" />
<property name="idCardNo" type="string" column="ID_CARD_NO"
not-null="true" />
</properties>
<property name="firstName" type="string" column="FIRST_NAME" />
<property name="lastName" type="string" column="LAST_NAME" />
<property name="email" type="string" column="EMAIL" />
<set name="addresses" table="CUSTOMER_ADDRESS" cascade="saveupdate,
delete">
<key column="CUSTOMER_ID" />
<many-to-many column="ADDRESS_ID" class="Address" />
</set>
</class>
</hibernate-mapping>
```

Now the <many-to-many> association from customer to address has been done. But it is only unidirectional. To make it bi-directional, we add the opposite definitions in the Address end, using the same join table.

```
public class Address {
private Long id;
private String city;
private String street;
private String doorplate;
private Set customers;
// Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Address" table="ADDRESS">
<id name="id" type="long" column="ID">
<generator class="native" />
</id>
<property name="city" type="string" column="CITY" />
<property name="street" type="string" column="STREET" />
<property name="doorplate" type="string" column="DOORPLATE" />
<set name="customers" table="CUSTOMER_ADDRESS">
<key column="ADDRESS_ID" />
<many-to-many column="CUSTOMER_ID" class="Customer" />
</set>
</class>
</hibernate-mapping>
```

But if you try to save this kind of object graph to the database, you will get an error. This is because Hibernate will save each side of the association in turn. For the Customer side association, several rows will be inserted into the CUSTOMER_ADDRESS table successfully. But for the Address side association, the same rows will be inserted into the same table so that a unique constraint violation occurred. To avoid saving the same association for two times, we can mark either side of the association as "inverse". Hibernate will ignore this side of association when saving the object.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Address" table="ADDRESS">
...
<set name="customers" table="CUSTOMER_ADDRESS" inverse="true">
<key column="ADDRESS_ID" />
<many-to-many column="CUSTOMER_ID" class="Customer" />
</set>
</class>
</hibernate-mapping>
```

4. Using a join table for one-to-many association

Remember that we can use a join table for a many-to-one association. Actually, we can use it for a one-to-many association as well. We can do it by using a <many-to-many> association type and marking it as unique="true".

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Book" table="BOOK">
...
<set name="chapters" table="BOOK_CHAPTER"
cascade="save-update,delete,delete-orphan">
<key column="BOOK_ID" />
<many-to-many column="CHAPTER_ID" class="Chapter" unique="true" />
</set>
</class>
</hibernate-mapping>
```

If we want to make a bi-directional one-to-many/many-to-one association using a join table, we can just define the many-to-one end in the same way as before. An important thing to notice is that we should mark either end of the bi-directional association as inverse. This time we choose the Chapter end as inverse, but it is also ok to choose another end.

```
<hibernate-mapping package="com.metaarchit.bookshop">
<class name="Chapter" table="CHAPTER">
...
<join table="BOOK_CHAPTER" optional="true" inverse="true">
<key column="CHAPTER_ID" unique="true" />
<many-to-one name="book" class="Book" column="BOOK_ID" not-null="true" />
</join>
</class>
</hibernate-mapping>
```