Hibernate Entity Relationships

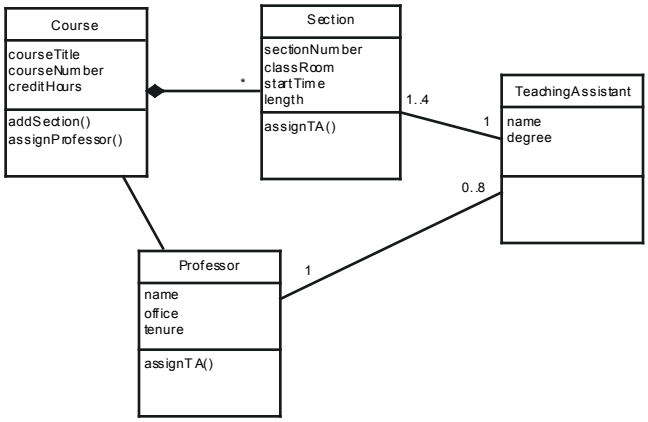
- Cardinality
- Navigability
- Lazy Instantation
- Cascading

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Relationships

- Object-oriented analysis and design results in a object model that include **relationships** between objects
- Characteristics of relationships include **cardinality** (multiplicity) and **navigability** (direction)
- With Hibernate, we need to persist the relationships at the database level

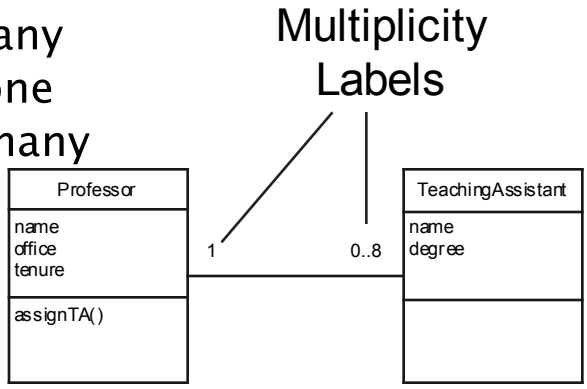


OO/AD developers often use the Unified Modeling Language (UML) to create diagrams that model the classes and their relationships.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Relationship Cardinality

- To implement a relationship, objects of the "source" class maintains a reference or list of references to the "target"
- **Cardinality** or multiplicity refers to how many references the source object maintains
- There are four common types:
  - One to one
  - One to many
  - Many to one
  - Many to many



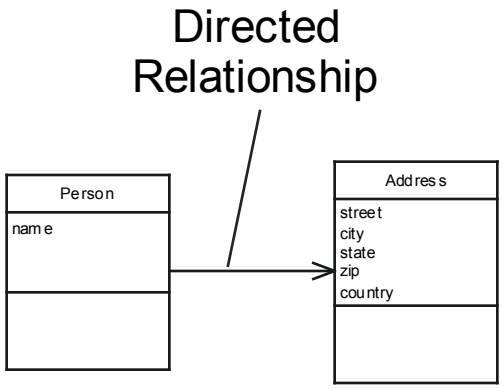
In a UML class diagram, multiplicity labels are an optional annotation that let you model the count of references on each side of the relationship. In this case, each TeachingAssistant "belongs" to a single Professor, and each Professor can mentor up to eight TeachingAssistants.

In a single-valued relationship, the source object maintains a single reference, while in a multi-valued relationship, the source must use an array or collection. In Hibernate, we generally use collections to implement multi-valued relationships.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Relationship Direction

- **Navigability** refers to whether the "source" object keeps a reference or list of references to the "target" object
- In a **unidirectional** or directed relationship, objects of one class can navigate to (i.e. call methods on) the related object, but not vice versa
- In a **bidirectional** relationship, navigation is possible in both directions



On a relationship line, an arrowhead indicates navigability from the source to the destination. Lack of any arrowheads implies navigability in both directions.

In this model, it's important to find an Address, given a Person, but it's not important to be able to find a Person given an Address.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Relationship Types

- Given four kinds of cardinality and two kinds of navigability, there are eight possible combinations, but only seven in practice
- OneToMany bidirectional is the same as ManyToOne bidirectional

```
1. OneToOne,   Unidirectional
2. OneToOne,   Bidirectional
3. OneToMany,  Unidirectional
4. ManyToOne,  Unidirectional
5. OneToMany,  Bidirectional
6. ManyToMany, Unidirectional
7. ManyToMany, Bidirectional
```


In addition to these seven combinations, Hibernate provides different database implementation strategies for some of them, including:

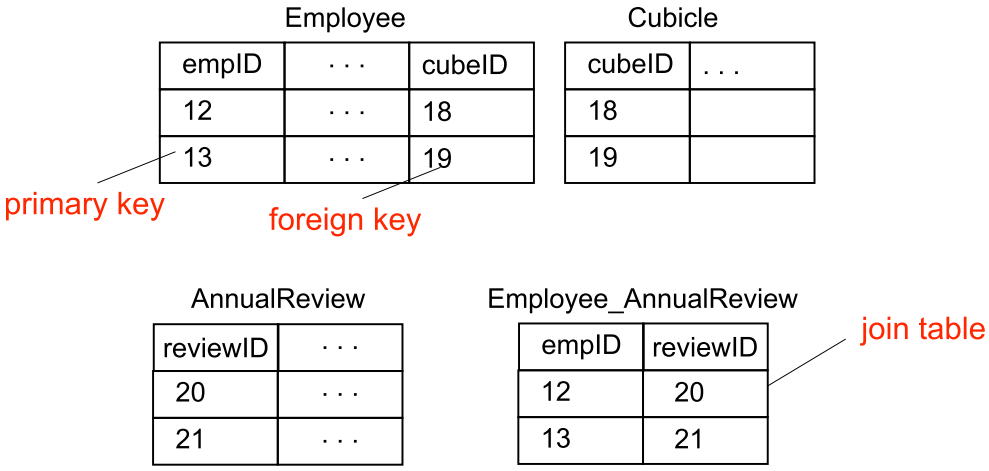
```
Foreign keys
Primary keys
Join tables
```

We will not cover all of the possible implementation strategies here, only the most common ones.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Relationships to the Database

- Hibernate defines strategies on how relationships are mapped to the database, including the use of **foreign keys** and **join tables**

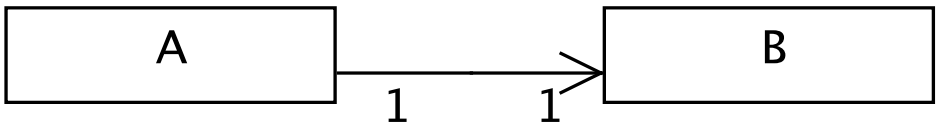


If you use a top-down approach, Hibernate will generate the table schema . If you use a bottom-up approach, you will likely need to configure the mappings to use your existing schema.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-One Unidirectional

- An 'A' object can reference a 'B' object, but not vice versa
- An 'A' object references at most one 'B' object
- A given 'B' object is referenced by at most one 'A' object
- Example: A - Employee, B - TravelProfile



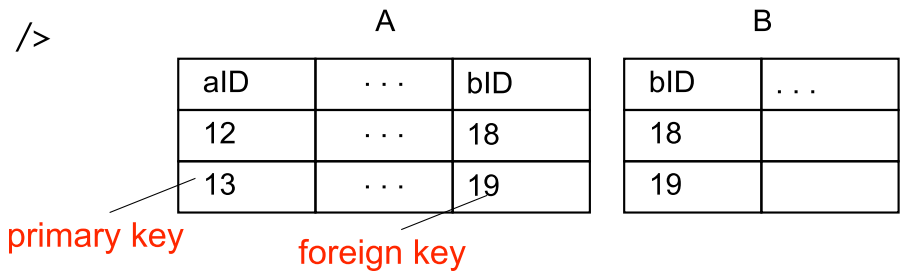
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-One Unidirectional Mapping

- The common mapping strategy for this relationship uses a foreign key in the "owning" table to the non-owning table

```
public class A
{
  private B b;
  . . .
}
<class name="A">
  . . .
  <many-to-one
    name="b"
    column="bID"
    unique="true" />
</class>
```

```
public class B
{
  . . .
}
<class name="B">
  . . .
</class>
```



Entity 'A' defines a reference to Entity 'B' so it can call methods, but 'B' maintains no reference to 'A'.

Entity 'A' is considered the owner, so its associated table contains a foreign key to the non-owning table, 'B'.

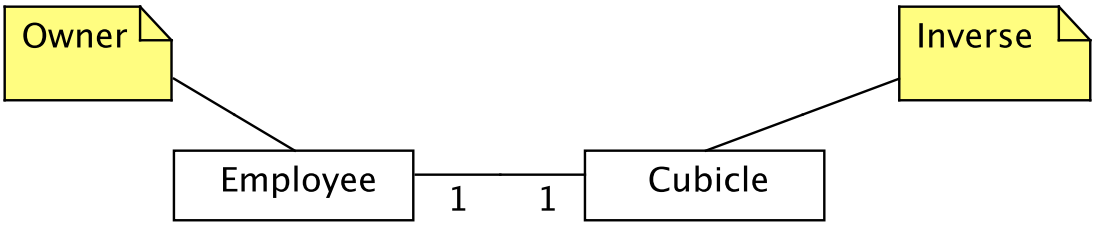
Notice the rather odd use of <many-to-one> in the mapping file for "A" -- that instructs Hibernate to use a foreign key mapping. The "unique" attribute causes Hibernate to generate a UNIQUE constraint on the foreign-key column so that all of the "A" rows have unique values. There's also a "not-null" attribute that causes Hibernate to make the relationship required - each "A" object MUST have a reference to a "B" object.

The other one-to-one unidirectional mapping strategy is to use the exact same key value for "B"'s primary key as "A"'s primary key. In that strategy, you can use the <one-to-one> XML mapping element. If you choose that strategy, then you must ensure that that related entities have the same key values. Hibernate provides the "foreign" primary-key generator for that purpose.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-One Bidirectional

- An 'A' object can reference a 'B' object, and a 'B' object can reference an 'A' object
- An 'A' object references at most one 'B' object
- A given 'B' object is referenced by at most one 'A' object
- A given 'A' object is referenced by at most one 'B' object
- Example: A - Employee, B - Cubicle



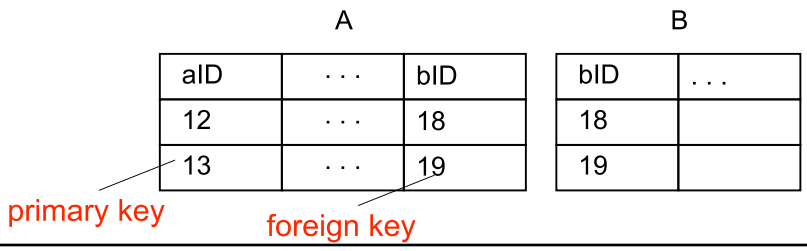
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-One Bidirectional Mapping

- In one mapping strategy, the "owning" entity's table will contain a foreign key to the inverse (non-owning) table

```
public class A
{
  private B b;
  . . .
}
<class name="A">
  . . .
  <many-to-one
    name="b"
    column="bID"
    unique="true"/>
</class>
```

```
public class B
{
  private A a;
  . . .
}
<class name="B">
  . . .
  <one-to-one name="a"
     property-ref="b"/>
</class>
```



Entity 'A' defines a reference to Entity 'B' so it can call methods, and 'B' also maintains a reference. The "property-ref" element is required to identify 'B' as the "inverse" side, i.e. the side whose table does NOT contain a foreign key.

Entity 'A' is the owner, so its associated table contains a foreign key to the non-owning table, 'B'.

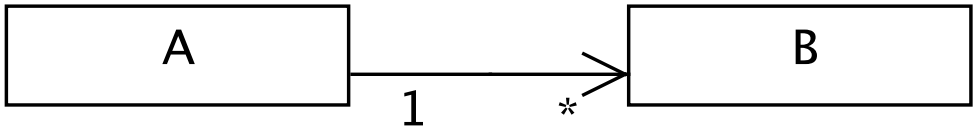
Again, note the odd usage: the owning class's mapping specifies <many-to-one> even though it's actually a one-to-one relationship - that's just how we tell Hibernate to use a foreign key.

The "inverse" side uses the more intuitive <one-to-one> mapping. Note that the property-ref attribute is not really necessary - one-to-one uses the primary key of the associated class by default.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-Many Unidirectional

- An 'A' object can reference multiple 'B' objects, but not vice versa
- A given 'B' object is referenced by at most one 'A' object
- Example: A - Employee, B - AnnualReview



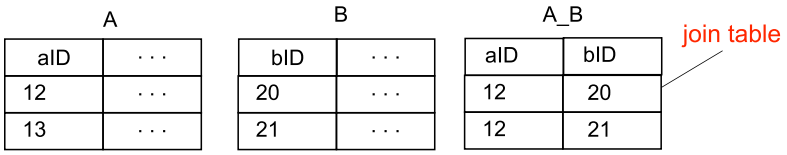
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-Many Unidirectional Mapping

- In the common mapping, there will be a **join table** that contains foreign keys of each entity table

```
public class A
{
  private Set<B> bset =
   new HashSet<B>();
  . . .
}
<class name="A">
  . . .
  <set name="bset"
    table="A_B">
    <key column="aID"/>
    <many-to-many
      column="bID"
      class="B"
      unique="true"/>
  </set>
</class>
2 - 12
```

```
public class B
{
  . . .
}
<class name="B">
  . . .
</class>
```



Entity 'A' defines a collection of references to Entity 'B' so it can call methods, but 'B' maintains no reference.

There is a join table that relates the entities in the common mapping.

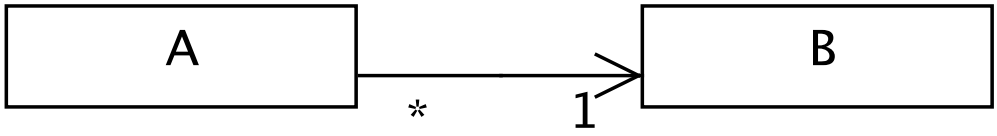
Note that you are not required to use Sets. Hibernate also supports lists and maps (e.g. you can use java.util.List in Java and <list> in the mapping). Also note that in the Java code, the reference to the collection is typed to an interface, not a concrete class - that's a Hibernate requirement. Also note that if we will create "A" objects and associate "B" objects BEFORE we persist the "A" objects, we need to instantiate the collection object explicitly as shown here. If we simply query, then Hibernate will instantiate a suitable concrete class for us.

If you'd prefer to use a simple foreign key relationship instead of a join table, use <one-to-many> with no "unique" attribute instead of <many-to-many>. Then the "B" table must contain a foreign-key column for the "A" primary key.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Many-to-One Unidirectional

- An 'A' object can reference a 'B' object, but not vice versa
- A given 'B' object can be referenced by multiple 'A' objects
- Example: A - Employee, B - Address



The difference between ManyToOne unidirectional and OneToOne unidirectional is that each target object can be referenced by multiple source objects.

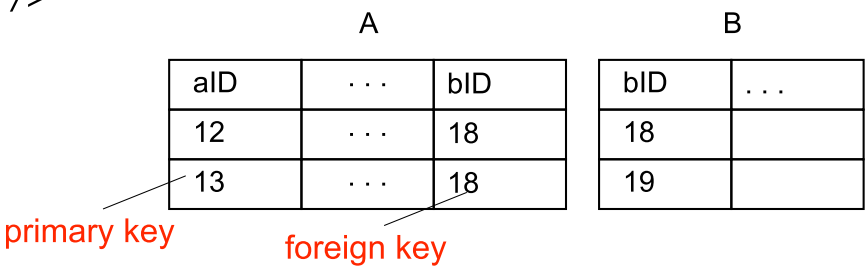
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Many-to-One Unidirectional Mapping

- In the common mapping, owning entity's table will contain a foreign key to the inverse (non-owning) table

```
public class A
{
  private B b;
  . . .
}
<class name="A">
  . . .
  <many-to-one
    name="b"
    column="bID" />
</class>
```

```
public class B
{
  . . .
}
<class name="B">
  . . .
</class>
```



Entity 'A' defines a reference to Entity 'B' so it can call methods, but 'B' maintains no reference to 'A'.

Entity 'A' is the owner, so its associated table contains a foreign key to the non-owning table, 'B'.

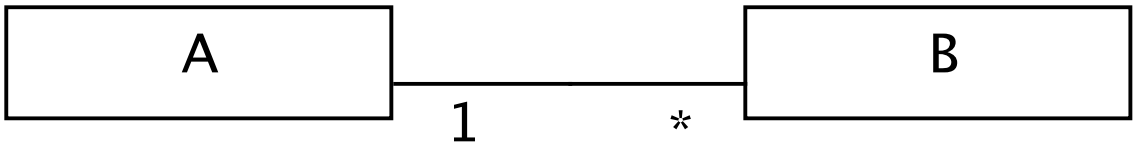
Note in the example tables that both entity 'A' rows reference the same entity 'B' row. That demonstrates the "many-to-one" semantics.

Also note that this is basically the same as one-to-one unidirectional, except that the owning class's mapping doesn't specify unique="true".

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-Many Bidirectional

- An 'A' object can reference multiple 'B' objects
- A 'B' object references at most a single 'A' object
- A given 'B' object is referenced by at most one 'A' object
- Example: A - Department, B - Employee



ManyToOne bidirectional is logically the same as this.

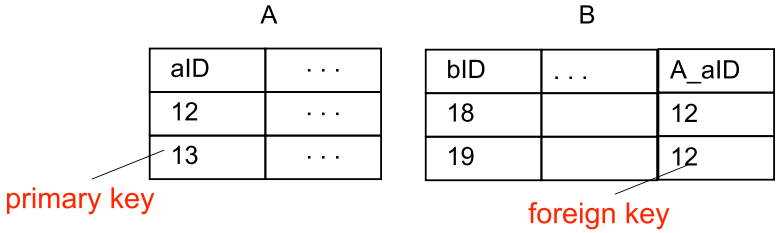
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

One-to-Many Bidirectional Mapping

- In the common mapping, the owning entity table contains a foreign key to the inverse entity

```
public class A
{
  private Set<B> bset =
   new HashSet<B>();
  . . .
}
<class name="A">
  . . .
  <set name="bset"
    inverse="true">
    <key column="A_aID"/>
    <one-to-many
      class="B" />
  </set>
</class>
```

```
public class B
{
  private A a;
  . . .
}
<class name="B">
  . . .
  <many-to-one name="a"
     column="A_aID"
     not-null="true"/>
</class>
```



Entity 'A' defines a collection of references to Entity 'B' so it can call methods, and 'B' maintains a reference to 'A'.

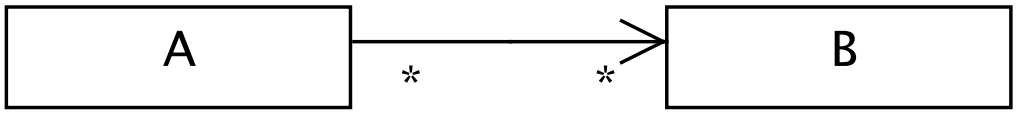
Entity 'B' is the owner, so its associated table contains a foreign key to the non-owning table, 'A'.

The "not-null" attribute in B's mapping is not strictly necessary - in a top-down design, it causes Hibernate to generate a NOT NULL constraint on the foreign-key column, ensuring that each B object has a related A object.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Many-to-Many Unidirectional

- An 'A' object can reference multiple 'B' objects, but not vice versa
- A given 'B' object can be referenced by multiple 'A' objects
- Example: A = Employee, B - Patent



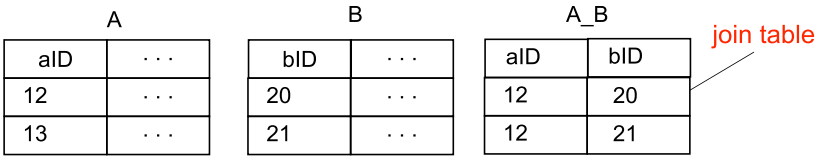
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Many-to-Many Unidirectional Mapping

- In the common mapping, there will be a **join table** that contains foreign keys of each entity table

```
public class A
{
  private Set<B> bset =
   new HashSet<B>();
  . . .
}
<class name="A">
  . . .
  <set name="bset"
    table="A_B">
    <key column="aID"/>
    <many-to-many
      column="bID"
      class="B" />
  </set>
</class>
```

```
public class B
{
  . . .
}
<class name="B">
  . . .
</class>
```



Entity 'A' defines a collection of references to Entity 'B' so it can call methods, but 'B' maintains no reference.

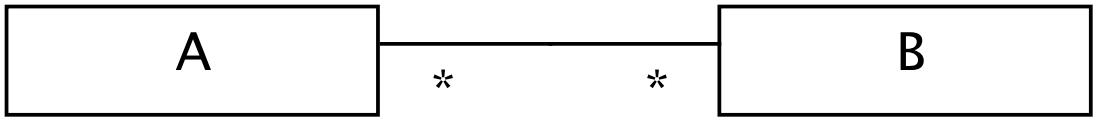
There is a join table that relates the entities.

The "A" class mapping is the same as the one-to-many unidirectional case, except we omit the unique="true" attribute.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Many-to-Many Bidirectional

- An 'A' object can reference multiple 'B' objects
- A 'B' object can reference multiple 'A' objects



Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Many-to-Many Bidirectional Mapping

- In the common mapping, there will be a **join table** that contains foreign keys of each entity table

```
public class A
{
  private Set<B> bset =
   new HashSet<B>();
  . . .
}
<class name="A">
  . . .
  <set name="bset"
    table="A_B">
    <key column="aID"/>
    <many-to-many
      column="bID"
      class="B" />
  </set>
</class>
```

```
public class B
{
  private Set<A> aset =
   new HashSet<A>();
  . . .
}
<class name="B">
  . . .
  <set name="aset"
    inverse="true"
    table="A_B">
    <key column="bID"/>
    <many-to-many
      column="aID"
      class="A" />
  </set>
</class>
```


Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Eager vs Lazy Fetching

- With **eager fetching**, the persistence provider retrieves all related objects when the relationship is established
- With **lazy fetching**, the persistence provider defers retrieving related objects until the object or collection is accessed
- Hibernate uses lazy fetching by default

In general, LAZY fetching is more efficient, since it defers transfers from the database until the related objects are accessed.

You can configure how fetching works in the mapping file and then override that configuration if you wish by writing explicit queries.

Behind the scenes, Hibernate uses "proxies" to implement lazy fetching -- at creation, Hibernate substitutes a proxy for the related object, then when the program uses the proxy, Hibernate can go fetch the actual data.

Older versions of Hibernate used eager as the default, but Hibernate 3 and later default to lazy, which is actually more sensible for most applications.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Eager vs Lazy Fetching, Single Value

- Suppose entity A has a single-valued association with entity B
- For eager fetching to work, entity B must be configured to allow it

```
public class A
{
  private B b;
. . .
}
<class name="A">
. . .
  <many-to-one
    name="b"
    column="bID"
    unique="true" />
</class>
```

```
public class B
{
. . .
}
<class name="B"
   lazy="false">
. . .
</class>
```

When the program creates an instance of "A", Hibernate will also fetch the actual entity "B", instead of defering fetching until the program actually accesses the reference to "B".

By marking the class itself as eager (not lazy), really you are telling Hibernate NOT to proxy the entity, so that means that ALL queries for this class will be executed eagerly. Another option is to specify lazy="false" on the many-to-one configuration in class A - then only that relationship is eager.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Eager vs Lazy Fetching, Multi Values

- Suppose entity A has a multi-valued association (collection) with entity B
- In A's mapping file, you can configure the collection as eager

```
public class A
{
  private Set<B> bset =
   new HashSet<B>();
  . . .
}
<class name="A">
  . . .
  <set name="bset"
    lazy="false"
    inverse="true"/>
    <key column="aID"/>
    <one-to-many class="B" />
  </set>
</class>
```

```
public class B
{
. . .
}
<class name="B">
. . .
</class>
```

Since it's actually the collection that's fetched lazily, entity B needs no special configuration.

Most Hibernate practitioners recommend using lazily instantiated collections as a general practice, using eager if you are sure that the application will access the related objects.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Detached Objects and Lazy Fetching

- If an object containing lazily-fetched relationships becomes detached, Hibernate will throw a LazyInitializationException if the application attempts to access the related object or collection
- This is especially an issue for Web applications - suppose a business tier retrieves entities, closes the Hibernate session, the returns the detached objects to Web tier for display

The problem with Web apps is that the business tier might return a lazily instantiated collection to the Web tier (servlet or JSP) for display, but when the Web tier iterates the collection, it receives the LIE (lazy initialization exception).

Two possible solutions:

1. Use eager fetching, which is inefficient if the Web page doesn't actually reference the related objects

2. Use a servlet filter to open and close the Hibernate session instead of doing that in the business tier

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Cascading

- Hibernate provides options on what the persistence manager does to related objects if you operate on the "parent" object
- By default, there is no cascading

```
public class A
{
  private Set<B> bset =
   new HashSet<B>();
  . . .
}
```

```
<class name="A">
  . . .
  <set name="bset"
    cascasde="all-delete-orphan"
    inverse="true"/>
. . .
  </set>
</class>
```

```
Cascade Options
---------------
none
save-update
delete
all
all-delete-orphan
delete-orphan
```

The "save-update" option tells Hibernate to automatically persist related objects when you call session.save() on the parent object.

The "delete" option tells Hibernate to delete related objects when you call session.delete() on the parent object.

The "delete-orphan" object tells Hibernate to detect when a "child" object is removed from an association or collection and automatically delete it from the database when you call saveOrUpdate() on the "parent" object.

The 'all' cascade type combines the save-update and delete options.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the different types of relationships and how Hibernate persists them

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum