6 Mapping collections and entity associations

This chapter covers

- Basic collection mapping strategies
- Mapping collections of value types
- Mapping a parent/children entity relationship

Two important (and sometimes difficult to understand) topics didn't appear in the previous chapters: the mapping of collections, and the mapping of associations between entity classes.

Most developers new to Hibernate are dealing with collections and entity associations for the first time when they try to map a typical *parent/child relationship*. But instead of jumping right into the middle, we start this chapter with basic collection mapping concepts and simple examples. After that, you'll be prepared for the first collection in an entity association—although we'll come back to more complicated entity association mappings in the next chapter. To get the full picture, we recommend you read both chapters.

6.1 Sets, bags, lists, and maps of value types

An object of *value type* has no database identity; it belongs to an entity instance, and its persistent state is embedded in the table row of the owning entity—at least, if an entity has a reference to a single instance of a valuetype. If an entity class has a collection of value types (or a collection of references to value-typed instances), you need an additional table, the so-called collection table.

Before you map collections of value types to collection tables, remember that value-typed classes don't have identifiers or identifier properties. The lifespan of a value-type instance is bounded by the lifespan of the owning entity instance. A value type doesn't support shared references.

Java has a rich collection API, so you can choose the collection interface and implementation that best fits your domain model design. Let's walk through the most common collection mappings.

Suppose that sellers in CaveatEmptor are able to attach images to `Item`s. An image is accessible only via the containing item; it doesn't need to support associations from any other entity in your system. The application manages the collection of images through the `Item` class, adding and removing elements. An image object has no life outside of the collection; it's dependent on an `Item` entity.

In this case, it isn't unreasonable to model the image class as a value type. Next. you need to decide what collection to use.

6.1.1 Selecting a collection interface

The idiom for a collection property in the Java domain model is always the same:

```
private <<Interface>> images = new <<Implementation>>();

...
// Getter and setter methods
```

Use an interface to declare the type of the property, not an implementation. Pick a matching implementation, and initialize the collection right away; doing so avoids uninitialized collections (we don't recommend initializing collections late, in constructors or setter methods).

If you work with JDK 5.0, you'll likely code with the generic versions of the JDK collections. Note that this isn't a requirement; you can also specify the contents of the collection explicitly in mapping metadata. Here's a typical generic `Set` with a type parameter:

```
private Set<String> images = new HashSet<String>();
...
// Getter and setter methods
```

Out of the box, Hibernate supports the most important JDK collection interfaces. In other words, it knows how to preserve the semantics of JDK collections, maps, and arrays in a persistent fashion. Each interface has a matching implementation supported by Hibernate, and it's important that you use the right combination. Hibernate only *wraps* the collection object you've already initialized on declaration of the field (or sometimes replaces it, if it's not the right one).

Without extending Hibernate, you can choose from the following collections:

- A `java.util.Set` is mapped with a `<set>` element. Initialize the collection with a `java.util.HashSet`. The order of its elements isn't preserved, and duplicate elements aren't allowed. This is the most common persistent collection in a typical Hibernate application.
- A `java.util.SortedSet` can be mapped with `<set>`, and the `sort` attribute can be set to either a comparator or natural ordering for in-memory sorting. Initialize the collection with a `java.util.TreeSet` instance.
- A `java.util.List` can be mapped with `<list>`, preserving the position of each element with an additional index column in the collection table. Initialize with a `java.util.ArrayList`.
- A `java.util.Collection` can be mapped with `<bag>` or `<idbag>`. Java doesn't have a `Bag` interface or an implementation; however, `java.util.Collection` allows bag semantics (possible duplicates, no element order is preserved). Hibernate supports persistent bags (it uses lists internally but ignores the index of the elements). Use a `java.util.ArrayList` to initialize a bag collection.
- A `java.util.Map` can be mapped with `<map>`, preserving key and value pairs. Use a `java.util.HashMap` to initialize a property.

- A `java.util.SortedMap` can be mapped with `<map>` element, and the `sort` attribute can be set to either a comparator or natural ordering for in-memory sorting. Initialize the collection with a `java.util.TreeMap` instance.
- Arrays are supported by Hibernate with `<primitive-array>` (for Java primitive value types) and `<array>` (for everything else). However, they're rarely used in domain models, because Hibernate can't wrap array properties. You lose lazy loading without bytecode instrumentation, and optimized dirty checking, essential convenience and performance features for persistent collections.

The JPA standard doesn't name all these options. The possible standard collection property types are `Set`, `List`, `Collection`, and `Map`. Arrays aren't considered.

Furthermore, the JPA specification only specifies that collection properties hold references to entity objects. Collections of value types, such as simple `String` instances, aren't standardized. However, the specification document already mentions that future versions of JPA will support collection elements of embeddable classes (in other words, value types). You'll need vendor-specific support if you want to map collections of value types with annotations. Hibernate Annotations include that support, and we'd expect many other JPA vendors support the same.

If you want to map collection interfaces and implementations not directly supported by Hibernate, you need to tell Hibernate about the semantics of your custom collections. The extension point in Hibernate is called `PersistentCollection`; usually you extend one of the existing `PersistentSet`, `PersistentBag`, or `PersistentList` classes. Custom persistent collections are not very easy to write and we don't recommend doing this if you aren't an experienced Hibernate user. An example can be found in the Hibernate test suite source code, as part of your Hibernate download package.

We now go through several scenarios, always implementing the collection of item images. You map it first in XML and then with Hibernate's support for collection annotations. For now, assume that the image is stored somewhere on the filesystem and that you keep just the filename in the database. How images are stored and loaded with this approach isn't discussed; we focus on the mapping.

6.1.2 Mapping a set

The simplest implementation is a `Set` of `String` image filenames. First, add a collection property to the `Item` class:

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}
public void setImages(Set images) {
    this.images = images;
}
```

Now, create the following mapping in the `Item`'s XML metadata:

```
<set name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

The image filenames are stored in a table named `ITEM_IMAGE`, the collection table. From the point of view of the database, this table is a separate entity, a separate table, but Hibernate hides this for you. The `<key>` element declares the foreign key column in the collection table that references the primary key `ITEM_ID` of the owning entity. The `<element>` tag declares this collection as a collection of value type instances—in this case, of strings.

A set can't contain duplicate elements, so the primary key of the `ITEM_IMAGE` collection table is a composite of both columns in the `<set>` declaration: `ITEM_ID` and `FILENAME`. You can see the schema in figure 6.1.

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

Figure 6.1
Table structure and example data for a collection of strings

It doesn't seem likely that you would allow the user to attach the same image more than once, but let's suppose you did. What kind of mapping would be appropriate in that case?

6.1.3 Mapping an identifier bag

An unordered collection that permits duplicate elements is called a *bag*. Curiously, the Java Collections framework doesn't include a bag implementation. However, the `java.util.Collection` interface has bag semantics, so you only need a matching implementation. You have two choices:

- Write the collection property with the `java.util.Collection` interface, and, on declaration, initialize it with an `ArrayList` of the JDK. Map the collection in Hibernate with a standard `<bag>` or `<idbag>` element. Hibernate has a built-in `PersistentBag` that can deal with lists; however, consistent with the contract of a bag, it ignores the position of elements in the `ArrayList`. In other words, you get a persistent `Collection`.
- Write the collection property with the `java.util.List` interface, and, on declaration, initialize it with an `ArrayList` of the JDK. Map it like the previous option, but expose a different collection interface in the domain model class. This approach works but isn't recommended, because clients using this collection property may think the order of elements is always preserved, which isn't the case if it's mapped as a `<bag>` or `<idbag>`.

We recommend the first option. Change the type of `images` in the `Item` class from `Set` to `Collection`, and initialize it with an `ArrayList`:

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

Note that the setter method accepts a `Collection`, which can be anything in the JDK collection interface hierarchy. However, Hibernate is smart enough to replace this when persisting the collection. (It also relies on an `ArrayList` internally, like you did in the declaration of the field.)

You also have to modify the collection table to permit duplicate `FILENAME`s; the table needs a different primary key. An `<idbag>` mapping adds a surrogate key column to the collection table, much like the synthetic identifiers you use for entity classes:

```
<idbag name="images" table="ITEM_IMAGE">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	1	fooimage1.jpg
2	1	fooimage1.jpg
3	3	barimage1.jpg

Figure 6.2 A surrogate primary key allows duplicate bag elements.

In this case, the primary key is the generated `ITEM_IMAGE_ID`, as you can see in figure 6.2. Note that the `native` generator for primary keys isn't supported for `<idbag>` mappings; you have to name a concrete strategy. This usually isn't a problem, because real-world applications often use a customized identifier generator anyway. You can also isolate your identifier generation strategy with placeholders; see chapter 3, section 3.3.4.3, "Using placeholders."

Also note that the `ITEM_IMAGE_ID` column isn't exposed to the application in any way. Hibernate manages it internally.

A more likely scenario is one in which you wish to preserve the order in which images are attached to the `Item`. There are a number of good ways to do this; one way is to use a real list, instead of a bag.

6.1.4 Mapping a list

First, let's update the `Item` class:

```
private List images = new ArrayList();
...
public List getImages() {
    return this.images;
}

public void setImages(List images) {
    this.images = images;
}
```

A `<list>` mapping requires the addition of an *index column* to the collection table. The index column defines the position of the element in the collection. Thus, Hibernate is able to preserve the ordering of the collection elements. Map the collection as a `<list>`:

```
<list name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <list-index column="POSITION"/>
```



```
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

(There is also an `index` element in the XML DTD, for compatibility with Hibernate 2.x. The new `list-index` is recommended; it's less confusing and does the same thing.)

The primary key of the collection table is a composite of `ITEM_ID` and `POSITION`. Notice that duplicate elements (`FILENAME`) are now allowed, which is consistent with the semantics of a list, see figure 6.3.

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	POSITION	FILENAME
1	0	fooimage1.jpg
1	1	fooimage2.jpg
1	2	foomage3.jpg

Figure 6.3 The collection table preserves the position of each element.

The index of the persistent list starts at zero. You could change this, for example, with `<list-index base="1"...>` in your mapping. Note that Hibernate adds null elements to your Java list if the index numbers in the database aren't continuous.

Alternatively, you could map a Java array instead of a list. Hibernate supports this; an array mapping is virtually identical to the previous example, except with different element and attribute names (`<array>` and `<array-index>`). However, for reasons explained earlier, Hibernate applications rarely use arrays.

Now, suppose that the images for an item have user-supplied names in addition to the filename. One way to model this in Java is a map, with names as keys and filenames as values of the map.

6.1.5 Mapping a map

Again, make a small change to the Java class:

```
private Map images = new HashMap();
...
public Map getImages() {
    return this.images;
}

public void setImages(Map images) {
    this.images = images;
}
```

Mapping a `<map>` (pardon us) is similar to mapping a list.

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME
1	Image One	fooimage1.jpg
1	Image Two	fooimage2.jpg
1	Image Three	foomage3.jpg

Figure 6.4 Tables for a map, using strings as indexes and elements

```
<map name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The primary key of the collection table is a composite of `ITEM_ID` and `IMAGENAME`. The `IMAGENAME` column holds the keys of the map. Again, duplicate elements are allowed; see figure 6.4 for a graphical view of the tables.

This map is unordered. What if you want to always sort your map by the name of the image?

6.1.6 Sorted and ordered collections

In a startling abuse of the English language, the words *sorted* and *ordered* mean different things when it comes to Hibernate persistent collections. A *sorted collection* is sorted in memory using a Java comparator. An *ordered collection* is ordered at the database level using an SQL query with an `order by` clause.

Let's make the map of images a sorted map. First, you need to change the initialization of the Java property to a `java.util.TreeMap` and switch to the `java.util.SortedMap` interface:

```
private SortedMap images = new TreeMap();
...
public SortedMap getImages() {
    return this.images;
}

public void setImages(SortedMap images) {
    this.images = images;
}
```

Hibernate handles this collection accordingly, if you map it as sorted:


```
<map name="images"
        table="ITEM_IMAGE"
        sort="natural">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

By specifying `sort="natural"`, you tell Hibernate to use a `SortedMap` and to sort the image names according to the `compareTo()` method of `java.lang.String`. If you need some other sort algorithm (for example, reverse alphabetical order), you may specify the name of a class that implements `java.util.Comparator` in the `sort` attribute. For example:

```
<map name="images"
     table="ITEM_IMAGE"
     sort="auction.util.comparator.ReverseStringComparator">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

A `java.util.SortedSet` (with a `java.util.TreeSet` implementation) is mapped like this:

```
<set name="images"
     table="ITEM_IMAGE"
     sort="natural">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Bags may not be sorted (there is no `TreeBag`, unfortunately), nor may lists; the order of list elements is defined by the list index.

Alternatively, instead of switching to the `Sorted*` interfaces and the (`Tree*` implementations), you may want to work with a linked map and to sort elements on the database side, not in memory. Keep the `Map`/`HashMap` declaration in the Java class, and create the following mapping:

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">
```

```
    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The expression in the `order-by` attribute is a fragment of an SQL `order by` clause. In this case, Hibernate orders the collection elements by the `IMAGENAME` column in ascending order during loading of the collection. You can even include an SQL function call in the `order-by` attribute:

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="lower(FILENAME) asc">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

You can order by any column of the collection table. Internally, Hibernate uses a `LinkedHashMap`, a variation of a map that preserves the insertion order of key elements. In other words, the order that Hibernate uses to add the elements to the collection, during loading of the collection, is the iteration order you see in your application. The same can be done with a set: Hibernate internally uses a `LinkedHashSet`. In your Java class, the property is a regular `Set/HashSet`, but Hibernate's internal wrapping with a `LinkedHashSet` is again enabled with the `order-by` attribute:

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="FILENAME asc">

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

You can also let Hibernate order the elements of a bag for you during collection loading. Your Java collection property is either `Collection/ArrayList` or `List/ArrayList`. Internally, Hibernate uses an `ArrayList` to implement a bag that preserves insertion-iteration order:

```
<idbag name="images"
       table="ITEM_IMAGE"
       order-by="ITEM_IMAGE_ID desc">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```


The linked collections Hibernate uses internally for sets and maps are available only in JDK 1.4 or later; older JDKs don't come with a `LinkedHashMap` and `LinkedHashSet`. Ordered bags are available in all JDK versions; internally, an `ArrayList` is used.

In a real system, it's likely that you'll need to keep more than just the image name and filename. You'll probably need to create an `Image` class for this extra information. This is the perfect use case for a collection of components.

6.2 Collections of components

You could map `Image` as an entity class and create a one-to-many relationship from `Item` to `Image`. However, this isn't necessary, because `Image` can be modeled as a value type: Instances of this class have a dependent lifecycle, don't need their own identity, and don't have to support shared references.

As a value type, the `Image` class defines the properties `name`, `filename`, `sizeX`, and `sizeY`. It has a single association with its owner, the `Item` entity class, as shown in figure 6.5.

As you can see from the composition association style (the black diamond), `Image` is a component of `Item`, and `Item` is the entity that is responsible for the lifecycle of `Image` instances. The multiplicity of the association further declares this association as many-valued—that is, many (or zero) `Image` instances for the same `Item` instance.

Let's walk through the implementation of this in Java and through a mapping in XML.

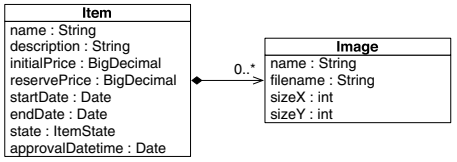

Figure 6.5 Collection of `Image` components in `Item`

6.2.1 Writing the component class

First, implement the `Image` class as a regular POJO. As you know from chapter 4, component classes don't have an identifier property. You must implement `equals()` (and `hashCode()`) and compare the `name`, `filename`, `sizeX`, and `sizeY` properties. Hibernate relies on this equality routine to check instances for modifications. A custom implementation of `equals()` and `hashCode()` isn't required for all component classes (we would have mentioned this earlier). However, we recommend it for any component class because the implementation is straightforward, and "better safe than sorry" is a good motto.

The `Item` class may have a `Set` of images, with no duplicates allowed. Let's map this to the database.

6.2.2 Mapping the collection

Collections of components are mapped similarly to collections of JDK value type. The only difference is the use of `<composite-element>` instead of an `<element>` tag. An ordered set of images (internally, a `LinkedHashSet`) can be mapped like this:

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">

    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <property name="name" column="IMAGENAME" not-null="true"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX" not-null="true"/>
       <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

The tables with example data are shown in figure 6.6.

This is a set, so the primary key of the collection table is a composite of the key column and all element columns: `ITEM_ID`, `IMAGENAME`, `FILENAME`, `SIZEX`, and `SIZEY`. Because these columns all appear in the primary key, you needed to declare them with `not-null="true"` (or make sure they're `NOT NULL` in any existing schema). No column in a composite primary key can be nullable—you can't identify what you don't know. This is probably a disadvantage of this particular mapping. Before you improve this (as you may guess, with an identifier bag), let's enable bidirectional navigation.

ITEM

ITEM_ID	ITEM_NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME	SIZEX	SIZEY
1	Foo	Foo.jpg	123	123
1	Bar	Bar.jpg	420	80
2	Baz	Baz.jpg	50	60

Figure 6.6
Example data tables for a collection of components mapping

6.2.3 Enabling bidirectional navigation

The association from `Item` to `Image` is unidirectional. You can navigate to the images by accessing the collection through an `Item` instance and iterating: `anItem.getImages().iterator()`. This is the only way you can get these image objects; no other entity holds a reference to them (value type again).

On the other hand, navigating from an image back to an item doesn't make much sense. However, it may be convenient to access a back pointer like `anImage.getItem()` in some cases. Hibernate can fill in this property for you if you add a `<parent>` element to the mapping:

```
<set name="images"
     table="ITEM_IMAGE"
     order-by="IMAGE_NAME asc">

    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <parent name="item"/>
       <property name="name" column="IMAGENAME" not-null="true"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX" not-null="true"/>
       <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>
```

True bidirectional navigation is impossible, however. You can't retrieve an `Image` independently and then navigate back to its parent `Item`. This is an important issue: You can load `Image` instances by querying for them. But these `Image` objects won't have a reference to their owner (the property is `null`) when you query in HQL or with a `Criteria`. They're retrieved as scalar values.

Finally, declaring all properties as `not-null` is something you may not want. You need a different primary key for the `IMAGE` collection table, if any of the property columns are nullable.

6.2.4 Avoiding not-null columns

Analogous to the additional surrogate identifier property an `<idbag>` offers, a surrogate key column would come in handy now. As a side effect, an `<idset>` would also allow duplicates—a clear conflict with the notion of a set. For this and other reasons (including the fact that nobody ever asked for this feature), Hibernate doesn't offer an `<idset>` or any surrogate identifier collection other than an `<idbag>`. Hence, you need to change the Java property to a `Collection` with bag semantics:

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

This collection now also allows duplicate `Image` elements—it's the responsibility of your user interface, or any other application code, to avoid these duplicate elements if you require set semantics. The mapping adds the surrogate identifier column to the collection table:

```
<idbag name="images"
       table="ITEM_IMAGE"
       order-by="IMAGE_NAME asc">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>

    <composite-element class="Image">
       <property name="name" column="IMAGENAME"/>
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX"/>
       <property name="sizeY" column="SIZEY"/>
    </composite-element>
</idbag>
```

The primary key of the collection table is now the `ITEM_IMAGE_ID` column, and it isn't important that you implement `equals()` and `hashCode()` on the `Image` class

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGENAME	FILENAME	SIZEX	SIZEY
1	1	Foo	Foo.jpg	123	123
2	1	Bar	Bar.jpg	420	80
3	2	Baz	Baz.jpg	NULL	NULL

Figure 6.7 Collection of `Image` components using a bag with surrogate key

(at least, Hibernate doesn't require it). Nor do you have to declare the properties with `not-null="true"`. They may be nullable, as can be seen in figure 6.7.

We should point out that there isn't a great deal of difference between this bag mapping and a standard parent/child entity relationship like the one you map later in this chapter. The tables are identical. The choice is mainly a matter of taste. A parent/child relationship supports shared references to the child entity and true bidirectional navigation. The price you'd pay is more complex lifecycles of objects. Value-typed instances can be created and associated with the persistent `Item` by adding a new element to the collection. They can be disassociated and permanently deleted by removing an element from the collection. If `Image` would be an entity class that supports shared references, you'd need more code in your application for the same operations, as you'll see later.

Another way to switch to a different primary key is a map. You can remove the `name` property from the `Image` class and use the image name as the key of a map:

```
<map name="images"
     table="ITEM_IMAGE"
     order-by="IMAGENAME asc">

    <key column="ITEM_ID"/>

    <map-key type="string" column="IMAGENAME"/>

    <composite-element class="Image">
       <property name="filename" column="FILENAME" not-null="true"/>
       <property name="sizeX" column="SIZEX"/>
       <property name="sizeY" column="SIZEY"/>
    </composite-element>
</map>
```

The primary key of the collection table is now a composite of `ITEM_ID` and `IMAGENAME`.

A composite element class like `Image` isn't limited to simple properties of basic type like `filename`. It may contain other components, mapped with `<nested-composite-element>`, and even `<many-to-one>` associations to entities. It can't

own collections, however. A composite element with a many-to-one association is useful, and we come back to this kind of mapping in the next chapter.

This wraps up our discussion of basic collection mappings in XML. As we mentioned at the beginning of this section, mapping collections of value types with annotations is different compared with mappings in XML; at the time of writing, it isn't part of the Java Persistence standard but is available in Hibernate.

6.3 Mapping collections with annotations

The Hibernate Annotations package supports nonstandard annotations for the mapping of collections that contain value-typed elements, mainly `org.hibernate.annotations.CollectionOfElements`. Let's walk through some of the most common scenarios again.

6.3.1 Basic collection mapping

The following maps a simple collection of `String` elements:

```
@org.hibernate.annotations.CollectionOfElements(
    targetElement = java.lang.String.class
)
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
private Set<String> images = new HashSet<String>();
```

The collection table `ITEM_IMAGE` has two columns; together, they form the composite primary key. Hibernate can automatically detect the type of the element if you use generic collections. If you don't code with generic collections, you need to specify the element type with the `targetElement` attribute—in the previous example it's therefore optional.

To map a persistent `List`, add `@org.hibernate.annotations.IndexColumn` with an optional base for the index (default is zero):

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
     name = "ITEM_IMAGE",
     joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.IndexColumn(
     name="POSITION", base = 1
)
@Column(name = "FILENAME")
private List<String> images = new ArrayList<String>();
```


If you forget the index column, this list would be treated as a bag collection, equivalent to a `<bag>` in XML.

For collections of value types, you'd usually use `<idbag>` to get a surrogate primary key on the collection table. A `<bag>` of value typed elements doesn't really work; duplicates would be allowed at the Java level, but not in the database. On the other hand, pure bags are great for one-to-many entity associations, as you'll see in chapter 7.

To map a persistent map, use `@org.hibernate.annotations.MapKey`:

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
     name = "ITEM_IMAGE",
     joinColumns = @JoinColumn(name = "ITEM_ID")
)
@org.hibernate.annotations.MapKey(
     columns = @Column(name="IMAGENAME")
)
@Column(name = "FILENAME")
private Map<String, String> images = new HashMap<String, String>();
```

If you forget the map key, the keys of this map would be automatically mapped to the column `MAPKEY`.

If the keys of the map are not simple strings but of an embeddable class, you can specify multiple map key columns that hold the individual properties of the embeddable component. Note that `@org.hibernate.annotations.MapKey` is a more powerful replacement for `@javax.persistence.MapKey`, which isn't very useful (see chapter 7, section 7.2.4 "Mapping maps").

6.3.2 Sorted and ordered collections

A collection can also be sorted or ordered with Hibernate annotations:

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.Sort(
    type = org.hibernate.annotations.SortType.NATURAL
)
private SortedSet<String> images = new TreeSet<String>();
```

(Note that without the `@JoinColumn` and/or `@Column`, Hibernate applies the usual naming conventions and defaults for the schema.) The `@Sort` annotation supports various `SortType` attributes, with the same semantics as the XML mapping options. The shown mapping uses a `java.util.SortedSet` (with a `java.util.TreeSet` implementation) and natural sort order. If you enable `SortType.COMPARATOR`, you also need to set the `comparator` attribute to a class that implements your comparison routine. Maps can also be sorted; however, as in XML mappings, there is no sorted Java bag or a sorted list (which has a persistent ordering of elements, by definition).

Maps, sets, and even bags, can be ordered on load, by the database, through an SQL fragment in the `ORDER BY` clause:

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@Column(name = "FILENAME", nullable = false)
@org.hibernate.annotations.OrderBy(
    clause = "FILENAME asc"
)
private Set<String> images = new HashSet<String>();
```

The `clause` attribute of the Hibernate-specific `@OrderBy` annotation is an SQL fragment that is passed on directly to the database; it can even contain function calls or any other native SQL keyword. See our explanation earlier for details about the internal implementation of sorting and ordering; the annotations are equivalent to the XML mappings.

6.3.3 Mapping a collection of embedded objects

Finally, you can map a collection of components, of user-defined value-typed elements. Let's assume that you want to map the same `Image` component class you've seen earlier in this chapter, with image names, sizes, and so on.

You need to add the `@Embeddable` component annotation on that class to enable embedding:

```
@Embeddable
public class Image {

    @org.hibernate.annotations.Parent
    Item item;

    @Column(length = 255, nullable = false)
    private String name;

    @Column(length = 255, nullable = false)
    private String filename;

    @Column(nullable = false)
    private int sizeX;

    @Column(nullable = false)
    private int sizeY;

    ... // Constructor, accessor methods, equals()/hashCode()
}
```


Note that you again map a back pointer with a Hibernate annotation; `anImage.getItem()` can be useful. You can leave out this property if you don't need this reference. Because the collection table needs all the component columns as the composite primary key, it's important that you map these columns as `NOT NULL`. You can now embed this component in a collection mapping and even override column definitions (in the following example you override the name of a single column of the component collection table; all others are named with the default strategy):

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@AttributeOverride(
    name = "element.name",
    column = @Column(name = "IMAGENAME",
                     length = 255,
                     nullable = false)
)
private Set<Image> images = new HashSet<Image>();
```

To avoid the non-nullable component columns you need a surrogate primary key on the collection table, like `<idbag>` provides in XML mappings. With annotations, use the `@CollectionId` Hibernate extension:

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "ITEM_IMAGE",
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@CollectionId(
    columns = @Column(name = "ITEM_IMAGE_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = "sequence"
)
private Collection<Image> images = new ArrayList<Image>();
```

You've now mapped all the basic and some more complex collections with XML mapping metadata, and annotations. Switching focus, we now consider collections with elements that aren't value types, but references to other entity instances. Many Hibernate users try to map a typical parent/children entity relationship, which involves a collection of entity references.

6.4 Mapping a parent/children relationship

From our experience with the Hibernate user community, we know that the first thing many developers try to do when they begin using Hibernate is a mapping of a parent/children relationship. This is usually the first time you encounter collections. It's also the first time you have to think about the differences between entities and value types, or get lost in the complexity of ORM.

Managing the associations between classes and the relationships between tables is at the heart of ORM. Most of the difficult problems involved in implementing an ORM solution relate to association management.

You mapped relationships between classes of value type in the previous section and earlier in the book, with varying multiplicity of the relationship ends. You map a *one* multiplicity with a simple `<property>` or as a `<component>`. The *many* association multiplicity requires a collection of value types, with `<element>` or `<composite-element>` mappings.

Now you want to map one- and many-valued relationships between entity classes. Clearly, entity aspects such as *shared references* and *independent lifecycle* complicate this relationship mapping. We'll approach these issues step by step; and, in case you aren't familiar with the term *multiplicity*, we'll also discuss that.

The relationship we show in the following sections is always the same, between the `Item` and `Bid` entity classes, as can be seen in figure 6.8.

Memorize this class diagram. But first, there's something we need to explain up front.

If you've used EJB CMP 2.0, you're familiar with the concept of a managed association (or managed relationship). CMP associations are called container managed relationships (CMRs) for a reason. Associations in CMP are inherently bidirectional. A change made to one side of an association is instantly reflected at the other side. For example, if you call `aBid.setItem(anItem)`, the container automatically calls `anItem.getBids().add(aBid)`.

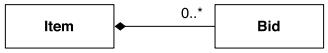

Figure 6.8
Relationship between `Item` and `Bid`

POJO-oriented persistence engines such as Hibernate don't implement managed associations, and POJO standards such as EJB 3.0 and Java Persistence don't require managed associations. Contrary to EJB 2.0 CMR, Hibernate and JPA associations are all inherently *unidirectional*. As far as Hibernate is concerned, the association from `Bid` to `Item` is a *different association* than the association from `Item` to `Bid`! This is a good thing—otherwise your entity classes wouldn't be usable outside of a runtime container (CMR was a major reason why EJB 2.1 entities were considered problematic).

Because associations are so important, you need a precise language for classifying them.

6.4.1 Multiplicity

In describing and classifying associations, we'll almost always use the term *multiplicity*. In our example, the multiplicity is just two bits of information:

- Can there be more than one `Bid` for a particular `Item`?
- Can there be more than one `Item` for a particular `Bid`?

After glancing at the domain model (see figure 6.8), you can conclude that the association from `Bid` to `Item` is a *many-to-one* association. Recalling that associations are directional, you classify the inverse association from `Item` to `Bid` as a *one-to-many* association.

There are only two more possibilities: *many-to-many* and *one-to-one*. We'll get back to these in the next chapter.

In the context of object persistence, we aren't interested in whether *many* means two or a maximum of five or unrestricted. And we're only barely interested in optionality of most associations; we don't especially care whether an associated instance is required or if the other end in an association can be `NULL` (meaning zero-to-many and to-zero association) However, these are important aspects in your relational data schema that influence your choice of integrity rules and the constraints you define in SQL DDL (see chapter 8, section 8.3, "Improving schema DDL").

6.4.2 The simplest possible association

The association from `Bid` to `Item` (and vice versa) is an example of the simplest possible kind of entity association. You have two properties in two classes. One is a collection of references, and the other a single reference.

First, here's the Java class implementation of `Bid`:

```
public class Bid {
    ...

    private Item item;

    public void setItem(Item item) {
        this.item = item;
    }

    public Item getItem() {
        return item;
    }

    ...
}
```

Next, this is the Hibernate mapping for this association:

```
<class
    name="Bid"
    table="BID">
    ...
    <many-to-one
        name="item"
        column="ITEM_ID"
        class="Item"
        not-null="true"/>

</class>
```

This mapping is called a *unidirectional many-to-one association*. (Actually, because it's unidirectional, you don't know what is on the other side, and you could just as well call this mapping a unidirectional to-one association mapping.) The column `ITEM_ID` in the `BID` table is a foreign key to the primary key of the `ITEM` table.

You name the class `Item`, which is the target of this association, explicitly. This is usually optional, because Hibernate can determine the target type with reflection on the Java property.

You added the `not-null` attribute because you can't have a bid without an item—a constraint is generated in the SQL DDL to reflect this. The foreign key column `ITEM_ID` in the `BID` can never be `NULL`, the association is not to-zero-or-one. The table structure for this association mapping is shown in figure 6.9.

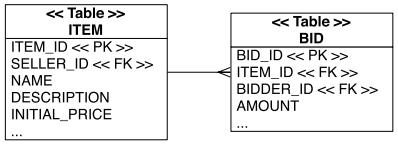

Figure 6.9
Table relationships and keys for a one-to-many mapping

In JPA, you map this association with the `@ManyToOne` annotation, either on the field or getter method, depending on the access strategy for the entity (determined by the position of the `@Id` annotation):

```
public class Bid {
    ...
    @ManyToOne( targetEntity = auction.model.Item.class )
    @JoinColumn(name = "ITEM_ID", nullable = false)
    private Item item;

    ...
}
```

There are two optional elements in this mapping. First, you don't have to include the `targetEntity` of the association; it's implicit from the type of the field. An explicit `targetEntity` attribute is useful in more complex domain models—for example, when you map a `@ManyToOne` on a getter method that returns a *delegate class*, which mimics a particular target entity interface.

The second optional element is the `@JoinColumn`. If you don't declare the name of the foreign key column, Hibernate automatically uses a combination of the target entity name and the database identifier property name of the target entity. In other words, if you don't add a `@JoinColumn` annotation, the default name for the foreign key column is `item` plus `id`, separated with an underscore. However, because you want to make the foreign key column `NOT NULL`, you need the annotation anyway to set `nullable = false`. If you generate the schema with the Hibernate Tools, the `optional="false"` attribute on the `@ManyToOne` would also result in a `NOT NULL` constraint on the generated column.

This was easy. It's critically important to realize that you can write a complete application without using anything else. (Well, maybe a shared primary key one-to-one mapping from time to time, as shown in the next chapter.) You don't need to map the other side of this class association, and you've already mapped everything present in the SQL schema (the foreign key column). If you need the `Item` instance for which a particular `Bid` was made, call `aBid.getItem()`, utilizing the entity association you created. On the other hand, if you need all bids that have been made for an item, you can write a query (in whatever language Hibernate supports).

One of the reasons you use a full object/relational mapping tool like Hibernate is, of course, that you don't want to write that query.

6.4.3 Making the association bidirectional

You want to be able to easily fetch all the bids for a particular item without an explicit query, by navigating and iterating through the network of persistent objects. The most convenient way to do this is with a collection property on `Item`: `anItem.getBids().iterator()`. (Note that there are other good reasons to map a collection of entity references, but not many. Always try to think of these kinds of collection mappings as a feature, not a requirement. If it gets too difficult, don't do it.)

You now map a collection of entity references by making the relationship between `Item` and `Bid` bidirectional.

First add the property and scaffolding code to the `Item` class:

```
public class Item {
    ...

    private Set bids = new HashSet();

    public void setBids(Set bids) {
        this.bids = bids;
    }

    public Set getBids() {
        return bids;
    }

    public void addBid(Bid bid) {
        bid.setItem(this);
        bids.add(bid);
    }

    ...
}
```

You can think of the code in `addBid()` (a convenience method) as implementing a managed association in the object model! (We had more to say about these methods in chapter 3, section 3.2, "Implementing the domain model." You may want to review the code examples there.)

A basic mapping for this one-to-many association looks like this:

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </set>

</class>
```


If you compare this with the collection mappings earlier in this chapter, you see that you map the content of the collection with a different element, `<one-to-many>`. This indicates that the collection contains not value type instances, but references to entity instances. Hibernate now knows how to treat shared references and the lifecycle of the associated objects (it disables all the implicit dependent lifecycle of value type instances). Hibernate also knows that the table used for the collection is the same table the target entity class is mapped to—the `<set>` mapping needs no `table` attribute.

The column mapping defined by the `<key>` element is the foreign key column `ITEM_ID` of the `BID` table, the same column you already mapped on the other side of the relationship.

Note that the table schema didn't change; it's the same as it was before you mapped the many side of the association. There is, however, one difference: The `not null="true"` attribute is missing. The problem is that you now have two different unidirectional associations mapped to the same foreign key column. What side controls that column?

At runtime, there are two different in-memory representations of the same foreign key value: the `item` property of `Bid` and an element of the `bids` collection held by an `Item`. Suppose the application modifies the association, by, for example, adding a bid to an item in this fragment of the `addBid()` method:

```
bid.setItem(item);
bids.add(bid);
```

This code is fine, but in this situation, Hibernate detects two changes to the in-memory persistent instances. From the point of view of the database, only one value has to be updated to reflect these changes: the `ITEM_ID` column of the `BID` table.

Hibernate doesn't transparently detect the fact that the two changes refer to the same database column, because at this point you've done nothing to indicate that this is a bidirectional association. In other words, you've mapped the same column twice (it doesn't matter that you did this in two mapping files), and Hibernate always needs to know about this because it can't detect this duplicate automatically (there is no reasonable default way it could be handled).

You need one more thing in the association mapping to make this a real bidirectional association mapping. The `inverse` attribute tells Hibernate that the collection is a mirror image of the `<many-to-one>` association on the other side:

```
<class
    name="Item"
    table="ITEM">
    ...

    <set name="bids"
         inverse="true">

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>

    </set>

</class>
```

Without the `inverse` attribute, Hibernate tries to execute two different SQL statements, both updating the same foreign key column, when you manipulate the link between two instances. By specifying `inverse="true"`, you explicitly tell Hibernate which end of the link it should not synchronize with the database. In this example, you tell Hibernate that it should propagate changes made at the `Bid` end of the association to the database, ignoring changes made only to the `bids` collection.

If you only call `anItem.getBids().add(bid)`, no changes are made persistent! You get what you want only if the other side, `aBid.setItem(anItem)`, is set correctly. This is consistent with the behavior in Java without Hibernate: If an association is bidirectional, you have to create the link with pointers on two sides, not just one. It's the primary reason why we recommend convenience methods such as `addBid()`—they take care of the bidirectional references in a system without container-managed relationships.

Note that an `inverse` side of an association mapping is always ignored for the generation of SQL DDL by the Hibernate schema export tools. In this case, the `ITEM_ID` foreign key column in the `BID` table gets a `NOT NULL` constraint, because you've declared it as such in the noninverse `<many-to-one>` mapping.

(Can you switch the inverse side? The `<many-to-one>` element doesn't have an `inverse` attribute, but you can map it with `update="false"` and `insert="false"` to effectively ignore it for any `UPDATE` or `INSERT` statements. The collection side is then noninverse and considered for insertion or updating of the foreign key column. We'll do this in the next chapter.)

Let's map this inverse collection side again, with JPA annotations:

```
public class Item {
    ...

    @OneToMany(mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...

}
```

The `mappedBy` attribute is the equivalent of the `inverse` attribute in XML mappings; however, it has to name the inverse property of the target entity. Note that you don't specify the foreign key column again here (it's mapped by the other side), so this isn't as verbose as the XML.

You now have a working *bidirectional* many-to-one association (which could also be called a bidirectional one-to-many association). One final option is missing if you want to make it a true parent/children relationship.

6.4.4 Cascading object state

The notion of a parent and a child implies that one takes care of the other. In practice, this means you need fewer lines of code to manage a relationship between a parent and a child, because some things can be taken care of automatically. Let's explore the options.

The following code creates a new `Item` (which we consider the parent) and a new `Bid` instance (the child):

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

session.save(newItem);
session.save(newBid);
```

The second call to `session.save()` seems redundant, if we're talking about a true parent/children relationship. Hold that thought, and think about entities and value types again: If both classes are entities, their instances have a completely independent lifecycle. New objects are transient and have to be made persistent if you want to store them in the database. Their relationship doesn't influence their lifecycle, if they're entities. If `Bid` would be a value type, the state of a `Bid` instance is the same as the state of its owning entity. In this case, however, `Bid` is a separate entity with its own completely independent state. You have three choices:

- Take care of the independent instances yourself, and execute additional `save()` and `delete()` calls on the `Bid` objects when needed—in addition to the Java code needed to manage the relationship (adding and removing references from collections, and so on).
- Make the `Bid` class a value type (a component). You can map the collection with a `<composite-element>` and get the implicit lifecycle. However, you lose other aspects of an entity, such as possible shared references to an instance.
- Do you need shared references to `Bid` objects? Currently, a particular `Bid` instance isn't referenced by more than one `Item`. However, imagine that a `User` entity also has a collection of `bids`, made by the user. To support shared references, you have to map `Bid` as an entity. Another reason you need shared references is the `successfulBid` association from `Item` in the full CaveatEmptor model. In this case, Hibernate offers *transitive persistence*, a feature you can enable to save lines of code and to let Hibernate manage the lifecycle of associated entity instances automatically.

You don't want to execute more persistence operations than absolutely necessary, and you don't want to change your domain model—you need shared references to `Bid` instances. The third option is what you'll use to simplify this parent/children example.

Transitive persistence

When you instantiate a new `Bid` and add it to an `Item`, the bid should become persistent automatically. You'd like to avoid making the `Bid` persistent explicitly with an extra `save()` operation.

To enable this transitive state across the association, add a `cascade` option to the XML mapping:

```
<class
    name="Item"
    table="ITEM">
    ...

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

        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>

    </set>

</class>
```


The `cascade="save-update"` attribute enables transitive persistence for `Bid` instances, if a particular `Bid` is referenced by a persistent `Item`, in the collection.

The `cascade` attribute is directional: It applies to only one end of the association. You could also add `cascade="save-update"` to the `<many-to-one>` association in the mapping of `Bid`, but because bids are created after items, doing so doesn't make sense.

JPA also supports cascading entity instance state on associations:

```
public class Item {
    ...

    @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE },
               mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...

}
```

Cascading options are *per operation* you'd like to be transitive. For native Hibernate, you cascade the `save` and `update` operations to associated entities with `cascade="save-update"`. Hibernate's object state management always bundles these two things together, as you'll learn in future chapters. In JPA, the (almost) equivalent operations are `persist` and `merge`.

You can now simplify the code that links and saves an `Item` and a `Bid`, in native Hibernate:

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

session.save(newItem);
```

All entities in the `bids` collection are now persistent as well, just as they would be if you called `save()` on each `Bid` manually. With the JPA `EntityManager` API, the equivalent to a `Session`, the code is as follows:

```
Item newItem = new Item();
Bid newBid = new Bid();

newItem.addBid(newBid); // Set both sides of the association

entityManager.persist(newItem);
```

Don't worry about the `update` and `merge` operations for now; we'll come back to them later in the book.

FAQ *What is the effect of `cascade` on `inverse`?* Many new Hibernate users ask this question. The answer is simple: The `cascade` attribute has nothing to do with the `inverse` attribute. They often appear on the same collection mapping. If you map a collection of entities as `inverse="true"`, you're controlling the generation of SQL for a bidirectional association mapping. It's a hint that tells Hibernate you mapped the same foreign key column twice. On the other hand, cascading is used as a convenience feature. If you decide to cascade operations from one side of an entity relationship to associated entities, you save the lines of code needed to manage the state of the other side manually. We say that object state becomes *transitive*. You can cascade state not only on collections of entities, but on all entity association mappings. `cascade` and `inverse` have in common the fact that they don't appear on collections of value types or on any other value-type mappings. The rules for these are implied by the nature of value types.

Are you finished now? Well, perhaps not quite.

Cascading deletion

With the previous mapping, the association between `Bid` and `Item` is fairly loose. So far, we have only considered making things persistent as a transitive state. What about deletion?

It seems reasonable that deletion of an item implies deletion of all bids for the item. In fact, this is what the composition (the filled out diamond) in the UML diagram means. With the current cascading operations, you have to write the following code to make that happen:

```
Item anItem = // Load an item

// Delete all the referenced bids
for ( Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext(); ) {

    Bid bid = it.next();

    it.remove();              // Remove reference from collection
    session.delete(bid);      // Delete it from the database
}

session.delete(anItem);       // Finally, delete the item
```

First you remove the references to the `bids` by iterating the collection. You delete each `Bid` instance in the database. Finally, the `Item` is deleted. Iterating and removing the references in the collection seems unnecessary; after all, you'll delete the `Item` at the end anyway. If you can guarantee that no other object (or row in any other table) holds a reference to these bids, you can make the deletion transitive.

Hibernate (and JPA) offer a cascading option for this purpose. You can enable cascading for the `delete` operation:

```
<set name="bids"
     inverse="true"
     cascade="save-update, delete">
...
```

The operation you cascade in JPA is called `remove`:

```
public class Item {
    ...
    @OneToMany(cascade = { CascadeType.PERSIST,
                           CascadeType.MERGE,
                           CascadeType.REMOVE },
               mappedBy = "item")
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

The same code to delete an item and all its bids is reduced to the following, in Hibernate or with JPA:

```
Item anItem = // Load an item
session.delete(anItem);
entityManager.remove(anItem);
```

The `delete` operation is now cascaded to all entities referenced in the collection. You no longer have to worry about removal from the collection and manually deleting those entities one by one.

Let's consider one further complication. You may have shared references to the `Bid` objects. As suggested earlier, a `User` may have a collection of references to the `Bid` instances they made. You can't delete an item and all its bids without removing these references first. You may get an exception if you try to commit this transaction, because a foreign key constraint may be violated.

You have to *chase the pointers*. This process can get ugly, as you can see in the following code, which removes all references from all users who have references before deleting the bids and finally the item:

```
Item anItem = // Load an item

// Delete all the referenced bids
for ( Iterator<Bid> it = anItem.getBids().iterator();
      it.hasNext(); ) {
```

```
    Bid bid = it.next();

    // Remove references from users who have made this bid
    Query q = session.createQuery(
      "from User u where :bid in elements(u.bids)"
    );
    q.setParameter("bid", bid);
    Collection usersWithThisBid = q.list();

    for (Iterator itUsers = usersWithThisBid.iterator();
         itUsers.hasNext();) {
         User user = (User) itUsers.next();
         user.getBids().remove(bid);
    }
}

session.delete(anItem);
// Finally, delete the item and the associated bids
```

Obviously, the additional query (in fact, many queries) isn't what you want. However, in a network object model, you don't have any choice other than executing code like this if you want to correctly set pointers and references—there is no persistent garbage collector or other automatic mechanism. No Hibernate cascading option helps you; you have to chase all references to an entity before you finally delete it.

(This isn't the whole truth: Because the `BIDDER_ID` foreign key column that represents the association from `User` to `Bid` is in the `BID` table, these references are automatically removed at the database level if a row in the `BID` table is deleted. This doesn't affect any objects that are already present in memory in the current unit of work, and it also doesn't work if `BIDDER_ID` is mapped to a different (intermediate) table. To make sure all references and foreign key columns are nulled out, you need to chase pointers in Java.)

On the other hand, if you don't have shared references to an entity, you should rethink your mapping and map the `bids` as a collection components (with the `Bid` as a `<composite-element>`). With an `<idbag>` mapping, even the tables look the same:

```
<class
    name="Item"
    table="ITEM">
    ...

    <idbag name="bids" table="BID">

        <collection-id type="long" column="BID_ID">
            <generator class="sequence"/>
        </collection-id>
```



```
        <key column="ITEM_ID" not-null="true"/>

        <composite-element class="Bid">
            <parent name="item"/>
            <property .../>
            ...
        </composite-element>

    </idbag>

</class>
```

The separate mapping for `Bid` is no longer needed.

If you really want to make this a one-to-many entity association, Hibernate offers another convenience option you may be interested in.

Enabling orphan deletion

The cascading option we explain now is somewhat difficult to understand. If you followed the discussion in the previous section, you should be prepared.

Imagine you want to delete a `Bid` from the database. Note that you aren't deleting the parent (the `Item`) in this case. The goal is to remove a row in the `BID` table. Look at this code:

```
anItem.getBids().remove(aBid);
```

If the collection has the `Bid` mapped as a collection of components, as in the previous section, this code triggers several operations:

- The `aBid` instance is removed from the collection `Item.bids`.
- Because `Bid` is mapped as a value type, and no other object can hold a reference to the `aBid` instance, the row representing this bid is deleted from the `BID` table by Hibernate.

In other words, Hibernate assumes that `aBid` is an orphan if it's removed from its owning entity's collection. No other in-memory persistent object is holding a reference to it. No foreign key value that references this row can be present in the database. Obviously, you designed your object model and mapping this way by making the `Bid` class an embeddable component.

However, what if `Bid` is mapped as an entity and the collection is a `<one-to-many>`? The code changes to

```
anItem.getBids().remove(aBid);
session.delete(aBid);
```

The `aBid` instance has its own lifecycle, so it can exist outside of the collection. By deleting it manually, you guarantee that nobody else will hold a reference to it,

and the row can be removed safely. You may have removed all other references manually. Or, if you didn't, the database constraints prevent any inconsistency, and you see a foreign key constraint exception.

Hibernate offers you a way to declare this guarantee for collections of entity references. You can tell Hibernate, "If I remove an element from this collection, it will be an entity reference, and it's going to be the only reference to that entity instance. You can safely delete it." The code that worked for deletion with a collection of components works with collections of entity references.

This option is called *cascade orphan delete*. You can enable it on a collection mapping in XML as follows:

```
<set name="bids"
     inverse="true"
     cascade="save-update, delete, delete-orphan">
...
```

With annotations, this feature is available only as a Hibernate extension:

```
public class Item {
    ...
    @OneToMany(cascade = { CascadeType.PERSIST,
                           CascadeType.MERGE,
                           CascadeType.REMOVE },
               mappedBy = "item")
    @org.hibernate.annotations.Cascade(
        value = org.hibernate.annotations.CascadeType.DELETE_ORPHAN
    )
    private Set<Bid> bids = new HashSet<Bid>();

    ...
}
```

Also note that this trick works only for collections of entity references in a one-to-many association; conceptually, no other entity association mapping supports it. You should ask yourself at this point, with so many cascading options set on your collection, whether a simple collection of components may be easier to handle. After all, you've enabled a dependent lifecycle for objects referenced in this collection, so you may as well switch to the implicit and fully dependent lifecycle of components.

Finally, let's look at the mapping in a JPA XML descriptor:

```
<entity-mappings>

    <entity class="auction.model.Item" access="FIELD">
        ...
        <one-to-many name="bids" mapped-by="item">
            <cascade>
                <cascade-persist/>
                <cascade-merge/>
                <cascade-remove/>
            </cascade>
        </one-to-many>
    </entity>

    <entity class="auction.model.Bid" access="FIELD">
        ...
        <many-to-one name="item">
            <join-column name="ITEM_ID"/>
        </many-to-one>
    </entity>

</entity-mappings>
```

Note that the Hibernate extension for cascade orphan deletion isn't available in this case.

6.5 Summary

You're probably a little overwhelmed by all the new concepts we introduced in this chapter. You may have to read it a few times, and we encourage you to try the code (and watch the SQL log). Many of the strategies and techniques we've shown in this chapter are key concepts of object/relational mapping. If you master collection mappings, and once you've mapped your first parent/children entity association, you'll have the worst behind you. You'll already be able to build entire applications!

Table 6.1 summarizes the differences between Hibernate and Java Persistence related to concepts discussed in this chapter.

Table 6.1 Hibernate and JPA comparison chart for chapter 6

Hibernate Core	Java Persistence and EJB 3.0
Hibernate provides mapping support for sets, lists, maps, bags, identifier bags, and arrays. All JDK collection interfaces are supported, and extension points for custom persistent collections are available.	Standardized persistent sets, lists, maps, and bags are supported.
Collections of value types and components are supported.	Hibernate Annotations is required for collections of value types and embeddable objects.
Parent/children entity relationships are supported, with transitive state cascading on associations per operation.	You can map entity associations and enable transitive state cascading on associations per operation.

Table 6.1 Hibernate and JPA comparison chart for chapter 6 *(continued)*

Hibernate Core	Java Persistence and EJB 3.0
Automatic deletion of orphaned entity instances is built in.	Hibernate Annotations is required for automatic deletion of orphaned entity instances.

We've covered only a tiny subset of the entity association options in this chapter. The remaining options we explore in detail in the next chapter are either rare or variations of the techniques we've just described.

7 *Advanced entity association mappings*

This chapter covers

- Mapping one-to-one and many-to-one entity associations
- Mapping one-to-many and many-to-many entity associations
- Polymorphic entity associations

When we use the word *associations*, we always refer to relationships between entities. In the previous chapter, we demonstrated a unidirectional many-to-one association, made it bidirectional, and finally turned it into a parent/children relationship (one-to-many and many-to-one with cascading options).

One reason we discuss more advanced entity mappings in a separate chapter is that quite a few of them are considered rare, or at least optional.

It's absolutely possible to only use component mappings and many-to-one (occasionally one-to-one) entity associations. You can write a sophisticated application without ever mapping a collection! Of course, efficient and easy access to persistent data, by iterating a collection for example, is one of the reasons why you use full object/relational mapping and not a simple JDBC query service. However, some exotic mapping features should be used with care and even *avoided* most of the time.

We'll point out recommended and optional mapping techniques in this chapter, as we show you how to map entity associations with all kinds of multiplicity, with and without collections.

7.1 Single-valued entity associations

Let's start with one-to-one entity associations.

We argued in chapter 4 that the relationships between `User` and `Address` (the user has a `billingAddress`, `homeAddress`, and `shippingAddress`) are best represented with a `<component>` mapping. This is usually the simplest way to represent one-to-one relationships, because the lifecycle is almost always dependent in such a case, it's either an aggregation or a composition in UML.

But what if you want a dedicated table for `Address`, and you map both `User` and `Address` as entities? One benefit of this model is the possibility for shared references—another entity class (let's say `Shipment`) can also have a reference to a particular `Address` instance. If a `User` has a reference to this instance, as their `shippingAddress`, the `Address` instance has to support shared references and needs its own identity.

In this case, `User` and `Address` classes have a true *one-to-one association*. Look at the revised class diagram in figure 7.1.

The first change is a mapping of the `Address` class as a stand-alone entity:

```
<class name="Address" table="ADDRESS">
    <id name="id" column="ADDRESS_ID">
        <generator .../>
    </id>
    <property name="street" column="STREET"/>
    <property name="city" column="CITY"/>
    <property name="zipcode" column="ZIPCODE"/>
</class>
```


We assume you won't have any difficulty creating the same mapping with annotations or changing the Java class to an entity, with an identifier property—this is the only change you have to make.

Now let's create the association mappings from other entities to that class. There are several choices, the first being a *primary key one-to-one* association.

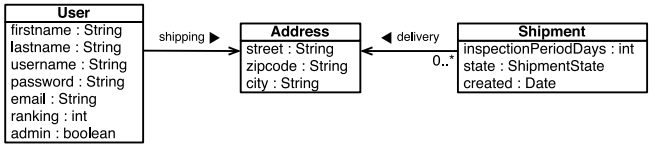

Figure 7.1 `Address` as an entity with two associations referencing the same instance

7.1.1 Shared primary key associations

Rows in two tables related by a primary key association share the same primary key values. The main difficulty with this approach is ensuring that associated instances are assigned the same primary key value when the objects are saved. Before we try to solve this problem, let's see how you map the primary key association.

Mapping a primary key association with XML

The XML mapping element that maps an entity association to a shared primary key entity is `<one-to-one>`. First you need a new property in the `User` class:

```
public class User {
   ...
   private Address shippingAddress;
   // Getters and setters
}
```

Next, map the association in `User.hbm.xml`:

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

You add a cascading option that is natural for this model: If a `User` instance is made persistent, you usually also want its `shippingAddress` to become persistent. Hence, the following code is all that is needed to save both objects:

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);

session.save(newUser);
```

Hibernate inserts a row into the `USERS` table and a row into the `ADDRESS` table. But wait, this doesn't work! How can Hibernate possibly know that the record in the `ADDRESS` table needs to get the same primary key value as the `USERS` row? At the beginning of this section, we intentionally didn't show you any primary-key generator in the mapping of `Address`.

You need to enable a special identifier generator.

The foreign identifier generator

If an `Address` instance is saved, it needs to get the primary key value of a `User` object. You can't enable a regular identifier generator, let's say a database sequence. The special `foreign` identifier generator for `Address` has to know where to get the right primary key value.

The first step to create this identifier binding between `Address` and `User` is a bidirectional association. Add a new `user` property to the `Address` entity:

```
public class Address {
   ...
    private User user;
   // Getters and setters
}
```

Map the new `user` property of an `Address` in Address.hbm.xml:

```
<one-to-one name="user"
            class="User"
            constrained="true"/>
```

This mapping not only makes the association bidirectional, but also, with `constrained="true"`, adds a foreign key constraint linking the primary key of the `ADDRESS` table to the primary key of the `USERS` table. In other words, the database guarantees that an `ADDRESS` row's primary key references a valid `USERS` primary key. (As a side effect, Hibernate can now also enable lazy loading of users when a shipping address is loaded. The foreign key constraint means that a user has to exist for a particular shipping address, so a proxy can be enabled without hitting the database. Without this constraint, Hibernate has to hit the database to find out if there is a user for the address; the proxy would then be redundant. We'll come back to this in later chapters.)

You can now use the special `foreign` identifier generator for `Address` objects:


```
<class name="Address" table="ADDRESS">

    <id name="id" column="ADDRESS_ID">
        <generator class="foreign">
            <param name="property">user</param>
        </generator>
    </id>
    ...
    <one-to-one name="user"
                class="User"
                constrained="true"/>

</class>
```

This mapping seems strange at first. Read it as follows: When an `Address` is saved, the primary key value is taken from the `user` property. The `user` property is a reference to a `User` object; hence, the primary key value that is inserted is the same as the primary key value of that instance. Look at the table structure in figure 7.2.

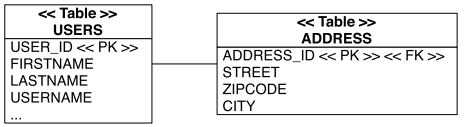

Figure 7.2
The `USERS` and `ADDRESS` tables have the same primary keys.

The code to save both objects now has to consider the bidirectional relationship, and it finally works:

```
User newUser = new User();
Address shippingAddress = new Address();

newUser.setShippingAddress(shippingAddress);
shippingAddress.setUser(newUser);            // Bidirectional

session.save(newUser);
```

Let's do the same with annotations.

Shared primary key with annotations

JPA supports one-to-one entity associations with the `@OneToOne` annotation. To map the association of `shippingAddress` in the `User` class as a shared primary key association, you also need the `@PrimaryKeyJoinColumn` annotation:

```
@OneToOne
@PrimaryKeyJoinColumn
private Address shippingAddress;
```

This is all that is needed to create a unidirectional one-to-one association on a shared primary key. Note that you need `@PrimaryKeyJoinColumns` (plural)

instead if you map with composite primary keys. In a JPA XML descriptor, a one-to-one mapping looks like this:

```
<entity-mappings>

    <entity class="auction.model.User" access="FIELD">
        ...
        <one-to-one name="shippingAddress">
            <primary-key-join-column/>
        </one-to-one>
    </entity>

</entity-mappings>
```

The JPA specification doesn't include a standardized method to deal with the problem of shared primary key generation, which means you're responsible for setting the identifier value of an `Address` instance correctly before you save it (to the identifier value of the linked `User` instance). Hibernate has an extension annotation for custom identifier generators which you can use with the `Address` entity (just like in XML):

```
@Entity
@Table(name = "ADDRESS")
public class Address {

    @Id @GeneratedValue(generator = "myForeignGenerator")
    @org.hibernate.annotations.GenericGenerator(
        name = "myForeignGenerator",
        strategy = "foreign",
        parameters = @Parameter(name = "property", value = "user")
    )
    @Column(name = "ADDRESS_ID")
    private Long id;

    ...
    private User user;
 }
```

Shared primary key one-to-one associations aren't uncommon but are relatively rare. In many schemas, a *to-one* association is represented with a foreign key field and a unique constraint.

7.1.2 One-to-one foreign key associations

Instead of sharing a primary key, two rows can have a foreign key relationship. One table has a foreign key column that references the primary key of the associated table. (The source and target of this foreign key constraint can even be the same table: This is called a *self-referencing* relationship.)

Let's change the mapping from a `User` to an `Address`. Instead of the shared primary key, you now add a `SHIPPING_ADDRESS_ID` column in the `USERS` table:

```
<class name="User" table="USERS">

    <many-to-one name="shippingAddress"
                 class="Address"
                 column="SHIPPING_ADDRESS_ID"
                 cascade="save-update"
                 unique="true"/>

</class>
```

The mapping element in XML for this association is `<many-to-one>`—not `<one-to-one>`, as you might have expected. The reason is simple: You don't care what's on the target side of the association, so you can treat it like a *to-one* association without the *many* part. All you want is to express "This entity has a property that is a reference to an instance of another entity" and use a foreign key field to represent that relationship. The database schema for this mapping is shown in figure 7.3.

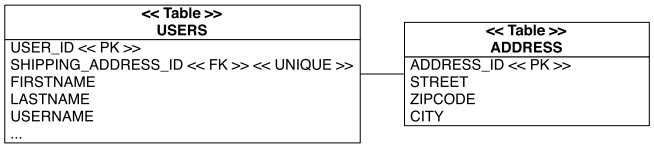

Figure 7.3 A one-to-one foreign key association between `USERS` and `ADDRESS`

An additional constraint enforces this relationship as a real *one* to one. By making the `SHIPPING_ADDRESS_ID` column `unique`, you declare that a particular address can be referenced by at most one user, as a shipping address. This isn't as strong as the guarantee from a shared primary key association, which allows a particular address to be referenced by at most one user, period. With several foreign key columns (let's say you also have unique `HOME_ADDRESS_ID` and `BILLING_ADDRESS_ID`), you can reference the same address target row several times. But in any case, two users can't share the same address for the same purpose.

Let's make the association from `User` to `Address` bidirectional.

Inverse property reference

The last foreign key association was mapped from `User` to `Address` with `<many-to-one>` and a `unique` constraint to guarantee the desired multiplicity. What mapping

element can you add on the `Address` side to make this association bidirectional, so that access from `Address` to `User` is possible in the Java domain model?

In XML, you create a `<one-to-one>` mapping with a property reference attribute:

```
<one-to-one name="user"
            class="User"
            property-ref="shippingAddress"/>
```

You tell Hibernate that the `user` property of the `Address` class is the inverse of a property on the other side of the association. You can now call `anAddress.getUser()` to access the user who's shipping address you've given. There is no additional column or foreign key constraint; Hibernate manages this pointer for you.

Should you make this association bidirectional? As always, the decision is up to you and depends on whether you need to navigate through your objects in that direction in your application code. In this case, we'd probably conclude that the bidirectional association doesn't make much sense. If you call `anAddress.getUser()`, you are saying "give me the user who has this address has its shipping address," not a very reasonable request. We recommend that a foreign key-based one-to-one association, with a unique constraint on the foreign key column—is almost always best represented without a mapping on the other side.

Let's repeat the same mapping with annotations.

Mapping a foreign key with annotations

The JPA mapping annotations also support a one-to-one relationship between entities based on a foreign key column. The main difference compared to the mappings earlier in this chapter is the use of `@JoinColumn` instead of `@PrimaryKeyJoinColumn`.

First, here's the to-one mapping from `User` to `Address` with the unique constraint on the `SHIPPING_ADDRESS_ID` foreign key column. However, instead of a `@ManyToOne` annotation, this requires a `@OneToOne` annotation:

```
public class User {
    ...

    @OneToOne
    @JoinColumn(name="SHIPPING_ADDRESS_ID")
    private Address shippingAddress;

    ...
}
```


Hibernate will now enforce the multiplicity with the unique constraint. If you want to make this association bidirectional, you need another `@OneToOne` mapping in the `Address` class:

```
public class Address {
    ...

    @OneToOne(mappedBy = "shippingAddress")
    private User user;

    ...
}
```

The effect of the `mappedBy` attribute is the same as the `property-ref` in XML mapping: a simple inverse declaration of an association, naming a property on the target entity side.

The equivalent mapping in JPA XML descriptors is as follows:

```
<entity-mappings>

    <entity class="auction.model.User" access="FIELD">
        ...
        <one-to-one name="shippingAddress">
            <join-column name="SHIPPING_ADDRESS_ID"/>
        </one-to-one>
    </entity>

    <entity class="auction.model.Address" access="FIELD">
        ...
        <one-to-one name="user" mapped-by="shippingAddress"/>
    </entity>

</entity-mappings>
```

You've now completed two basic single-ended association mappings: the first with a shared primary key, the second with a foreign key reference. The last option we want to discuss is a bit more exotic: mapping a one-to-one association with the help of an additional table.

7.1.3 *Mapping with a join table*

Let's take a break from the complex CaveatEmptor model and consider a different scenario. Imagine you have to model a data schema that represents an office allocation plan in a company. Common entities include people working at desks. It seems reasonable that a desk may be vacant and have no person assigned to it. On the other hand, an employee may work at home, with the same result. You're dealing with an *optional* one-to-one association between `Person` and `Desk`.

If you apply the mapping techniques we discussed in the previous sections, you may come to the following conclusions: `Person` and `Desk` are mapped to two tables, with one of them (let's say the `PERSON` table) having a foreign key column that references the other table (such as `ASSIGNED_DESK_ID`) with an additional unique constraint (so two people can't be assigned the same desk). The relationship is optional if the foreign key column is nullable.

On second thought, you realize that the assignment between persons and desks calls for another table that represents `ASSIGNMENT`. In the current design, this table has only two columns: `PERSON_ID` and `DESK_ID`. The multiplicity of these foreign key columns is enforced with a `unique` constraint on both—a particular person and desk can only be assigned once, and only one such an assignment can exist.

It also seems likely that one day you'll need to extend this schema and add columns to the `ASSIGNMENT` table, such as the date when a person was assigned to a desk. As long as this isn't the case, however, you can use object/relational mapping to hide the intermediate table and create a one-to-one Java entity association between only two classes. (This situation changes completely once additional columns are introduced to `ASSIGNMENT`.)

Where does such an optional one-to-one relationship exist in CaveatEmptor?

The CaveatEmptor use case

Let's consider the `Shipment` entity in CaveatEmptor again and discuss its purpose. Sellers and buyers interact in CaveatEmptor by starting and bidding on auctions. The shipment of the goods seems to be outside the scope of the application; the seller and the buyer agree on a method of shipment and payment after the auction ends. They can do this offline, outside of CaveatEmptor. On the other hand, you could offer an extra *escrow service* in CaveatEmptor. Sellers would use this service to create a trackable shipment once the auction completed. The buyer would pay the price of the auction item to a trustee (you), and you'd inform the seller that the money was available. Once the shipment arrived and the buyer accepted it, you'd transfer the money to the seller.

If you've ever participated in an online auction of significant value, you've probably used such an escrow service. But you want more service in CaveatEmptor. Not only will you provide trust services for completed auctions, but you'll also allow users to create a trackable and trusted shipment for any deal they make outside an auction, outside CaveatEmptor.

This scenario calls for a `Shipment` entity with an optional one-to-one association to an `Item`. Look at the class diagram for this domain model in figure 7.4.

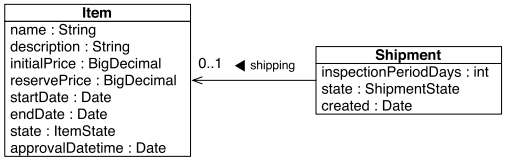

Figure 7.4 A shipment has an optional link with a single auction item.

In the database schema, you add an intermediate link table called `ITEM_SHIPMENT`. A row in this table represents a `Shipment` made in the context of an auction. The tables are shown in figure 7.5.

You now map two classes to three tables: first in XML, and then with annotations.

Mapping a join table in XML

The property that represents the association from `Shipment` to `Item` is called `auction`:

```
public class Shipment {

    ...
    private Item auction;
    ...
    // Getter/setter methods
}
```

Because you have to map this association with a foreign key column, you need the `<many-to-one>` mapping element in XML. However, the foreign key column isn't in the `SHIPMENT` table, it's in the `ITEM_SHIPMENT` join table. With the help of the `<join>` mapping element, you move it there.

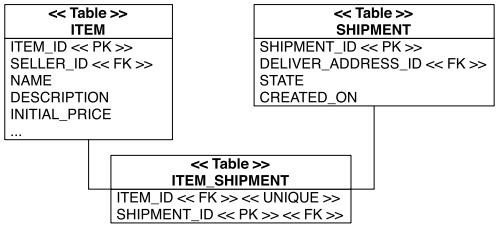

Figure 7.5 An optional one-to-many relationship mapped to a join table

```
<class name="Shipment" table="SHIPMENT">

    <id name="id" column="SHIPMENT_ID">...</id>

    ...

    <join table="ITEM_SHIPMENT" optional="true">
        <key column="SHIPMENT_ID"/>
        <many-to-one name="auction"
                     column="ITEM_ID"
                     not-null="true"
                     unique="true"/>
    </join>

</class>
```

The join table has two foreign key columns: `SHIPMENT_ID`, referencing the primary key of the `SHIPMENT` table; and `ITEM_ID`, referencing the `ITEM` table. The `ITEM_ID` column is unique; a particular item can be assigned to exactly one shipment. Because the primary key of the join table is `SHIPMENT_ID`, which makes this column also unique, you have a guaranteed one-to-one multiplicity between `Shipment` and `Item`.

By setting `optional="true"` on the `<join>` mapping, you tell Hibernate that it should insert a row into the join table only if the properties grouped by this mapping are non-null. But if a row needs to be inserted (because you called `aShipment.setAuction(anItem)`), the `NOT NULL` constraint on the `ITEM_ID` column applies.

You could map this association bidirectional, with the same technique on the other side. However, optional one-to-one associations are unidirectional most of the time.

JPA also supports association join tables as *secondary tables* for an entity.

Mapping secondary join tables with annotations

You can map an optional one-to-one association to an intermediate join table with annotations:

```
public class Shipment {

    @OneToOne
    @JoinTable(
        name="ITEM_SHIPMENT",
        joinColumns = @JoinColumn(name = "SHIPMENT_ID"),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID")
    )
    private Item auction;
    ...
    // Getter/setter methods
}
```


You don't have to specify the `SHIPMENT_ID` column because it's automatically considered to be the join column; it's the primary key column of the `SHIPMENT` table.

Alternatively, you can map properties of a JPA entity to more than one table, as demonstrated in "Moving properties into a secondary table" in chapter 8, section 8.1.3. First, you need to declare the secondary table for the entity:

```
@Entity
@Table(name = "SHIPMENT")
@SecondaryTable(name = "ITEM_SHIPMENT")
public class Shipment {

    @Id @GeneratedValue
    @Column(name = "SHIPMENT_ID")
    private Long id;

    ...
}
```

Note that the `@SecondaryTable` annotation also supports attributes to declare the foreign-key column name—the equivalent of the `<key column="..."/>` you saw earlier in XML and the `joinColumn(s)` in a `@JoinTable`. If you don't specify it, the primary-key column name of the entity is used—in this case, again `SHIPMENT_ID`.

The `auction` property mapping is a `@OneToOne`; and as before, the foreign key column referencing the `ITEM` table is moved to the intermediate secondary table:

```
...
public class Shipment {
    ...
    @OneToOne
    @JoinColumn(table = "ITEM_SHIPMENT", name = "ITEM_ID")
    private Item auction;
}
```

The table for the target `@JoinColumn` is named explicitly. Why would you use this approach instead of the (simpler) `@JoinTable` strategy? Declaring a secondary table for an entity is useful if not only one property (the many-to-one in this case) but several properties must be moved into the secondary table. We don't have a great example with `Shipment` and `Item`, but if your `ITEM_SHIPMENT` table would have additional columns, mapping these columns to properties of the `Shipment` entity might be useful.

This completes our discussion of one-to-one association mappings. To summarize, use a shared primary key association if one of the two entities seems more important and can act as the primary key source. Use a foreign key association in all other cases, and a hidden intermediate join table when your one-to-one association is optional.

We now focus on many-valued entity associations, including more options for one-to-many, and finally, many-to-many mappings.

7.2 Many-valued entity associations

A *many-valued* entity association is by definition a collection of entity references. You mapped one of these in the previous chapter, section 6.4, "Mapping a parent/children relationship." A parent entity instance has a collection of references to many child objects—hence, *one-to-many*.

One-to-many associations are the most important kind of entity association that involves a collection. We go so far as to discourage the use of more exotic association styles when a simple bidirectional many-to-one/one-to-many will do the job. A many-to-many association may always be represented as two many-to-one associations to an intervening class. This model is usually more easily extensible, so we tend not to use many-to-many associations in applications. Also remember that you don't have to map *any* collection of entities, if you don't want to; you can always write an explicit query instead of direct access through iteration.

If you decide to map collections of entity references, there are a few options and more complex situations that we discuss now, including a many-to-many relationship.

7.2.1 One-to-many associations

The parent/children relationship you mapped earlier was a bidirectional association, with a `<one-to-many>` and a `<many-to-one>` mapping. The *many* end of this association was implemented in Java with a `Set`; you had a collection of `bids` in the `Item` class.

Let's reconsider this mapping and focus on some special cases.

Considering bags

It's *possible* to use a `<bag>` mapping instead of a set for a bidirectional one-to-many association. Why would you do this?

Bags have the most efficient performance characteristics of all the collections you can use for a bidirectional one-to-many entity association (in other words, if the collection side is `inverse="true"`). By default, collections in Hibernate are loaded only when they're accessed for the first time in the application. Because a bag doesn't have to maintain the index of its elements (like a list) or check for duplicate elements (like a set), you can add new elements to the bag without triggering the loading. This is an important feature if you're going to map a possibly large collection of entity references. On the other hand, you can't eager-fetch two collections of bag type simultaneously (for example, if `bids` and `images` of an `Item` were one-to-many bags). We'll come back to fetching strategies in chapter 13, section 13.1, "Defining the global fetch plan." In general we would say that a bag is the best inverse collection for a one-to-many association.

To map a bidirectional one-to-many association as a bag, you have to replace the type of the `bids` collection in the `Item` persistent class with a `Collection` and an `ArrayList` implementation. The mapping for the association between `Item` and `Bid` is left essentially unchanged:

```
<class name="Bid"
       table="BID">
    ...
    <many-to-one name="item"
                 column="ITEM_ID"
                 class="Item"
                 not-null="true"/>

</class>
<class name="Item"
       table="ITEM">
    ...
    <bag name="bids"
         inverse="true">
        <key column="ITEM_ID"/>
        <one-to-many class="Bid"/>
    </bag>

</class>
```

You rename the `<set>` element to `<bag>`, making no other changes. Even the tables are the same: The `BID` table has the `ITEM_ID` foreign key column. In JPA, all `Collection` and `List` properties are considered to have bag semantics, so the following is equivalent to the XML mapping:

```
public class Item {
    ...

    @OneToMany(mappedBy = "item")
    private Collection<Bid> bids = new ArrayList<Bid>();

    ...

}
```

A bag also allows duplicate elements, which the set you mapped earlier didn't. It turns out that this isn't relevant in this case, because *duplicate* means you've added a particular reference to the same `Bid` instance several times. You wouldn't do this

in your application code. But even if you add the same reference several times to this collection, Hibernate ignores it—it's mapped inverse.

Unidirectional and bidirectional lists

If you need a real list to hold the position of the elements in a collection, you have to store that position in an additional column. For the one-to-many mapping, this also means you should change the `bids` property in the `Item` class to `List` and initialize the variable with an `ArrayList` (or keep the `Collection` interface from the previous section, if you don't want to expose this behavior to a client of the class).

The additional column that holds the position of a reference to a `Bid` instance is the `BID_POSITION`, in the mapping of `Item`:

```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>

</class>
```

So far this seems straightforward; you've changed the collection mapping to `<list>` and added the `<list-index>` column `BID_POSITION` to the collection table (which in this case is the `BID` table). Verify this with the table shown in figure 7.6.

This mapping isn't really complete. Consider the `ITEM_ID` foreign key column: It's `NOT NULL` (a bid has to reference an item). The first problem is that you don't specify this constraint in the mapping. Also, because this mapping is unidirectional (the collection is noninverse), you have to assume that there is no opposite side mapped to the same foreign key column (where this constraint could be declared). You need to add a `not-null="true"` attribute to the `<key>` element of the collection mapping:

BID

BID_ID	ITEM_ID	BID_POSITION	AMOUNT	CREATED_ON
1	1	0	99.00	19.04.08 23:11
2	1	1	123.00	19.04.08 23:12
3	2	0	433.00	20.04.08 09:30

Figure 7.6 Storing the position of each bid in the list collection


```
<class name="Item"
       table="ITEM">
    ...
    <list name="bids">
        <key column="ITEM_ID" not-null="true"/>
        <list-index column="BID_POSITION"/>
        <one-to-many class="Bid"/>
    </list>

</class>
```

Note that the attribute has to be on the `<key>` mapping, not on a possible nested `<column>` element. Whenever you have a noninverse collection of entity references (most of the time a one-to-many with a list, map, or array) and the foreign key join column in the target table is not nullable, you need to tell Hibernate about this. Hibernate needs the hint to order `INSERT` and `UPDATE` statements correctly, to avoid a constraint violation.

Let's make this bidirectional with an `item` property of the `Bid`. If you follow the examples from earlier chapters, you might want to add a `<many-to-one>` on the `ITEM_ID` foreign key column to make this association bidirectional, and enable `inverse="true"` on the collection. Remember that Hibernate ignores the state of an inverse collection! This time, however, the collection contains information that is needed to update the database correctly: the position of its elements. If only the state of each `Bid` instance is considered for synchronization, and the collection is inverse and ignored, Hibernate has no value for the `BID_POSITION` column.

If you map a bidirectional one-to-many entity association with an indexed collection (this is also true for maps and arrays), you have to switch the inverse sides. You can't make an indexed collection `inverse="true"`. The collection becomes responsible for state synchronization, and the *one* side, the `Bid`, has to be made inverse. However, there is no `inverse="true"` for a many-to-one mapping so you need to simulate this attribute on a `<many-to-one>`:

```
<class name="Bid"
       table="BID">
    ...
    <many-to-one name="item"
                 column="ITEM_ID"
                 class="Item"
                 not-null="true"
                 insert="false"
                 update="false"/>

</class>
```

Setting `insert` and `update` to `false` has the desired effect. As we discussed earlier, these two attributes used together make a property effectively *read-only*. This side of the association is therefore ignored for any write operations, and the state of the collection (including the index of the elements) is the relevant state when the in-memory state is synchronized with the database. You've switched the inverse/noninverse sides of the association, a requirement if you switch from a set or bag to a list (or any other indexed collection).

The equivalent in JPA, an indexed collection in a bidirectional one-to-many mapping, is as follows:

```
public class Item {
    ...

    @OneToMany
    @JoinColumn(name = "ITEM_ID", nullable = false)
    @org.hibernate.annotations.IndexColumn(name = "BID_POSITION")
    private List<Bid> bids = new ArrayList<Bid>();

    ...
}
```

This mapping is noninverse because no `mappedBy` attribute is present. Because JPA doesn't support persistent indexed lists (only *ordered* with an `@OrderBy` at load time), you need to add a Hibernate extension annotation for index support. Here's the other side of the association in `Bid`:

```
public class Bid {
    ...

    @ManyToOne
    @JoinColumn(name = "ITEM_ID", nullable = false,
                updatable = false, insertable = false)
    private Item item;

    ...
}
```

We now discuss one more scenario with a one-to-many relationship: an association mapped to an intermediate join table.

Optional one-to-many association with a join table

A useful addition to the `Item` class is a `buyer` property. You can then call `anItem.getBuyer()` to access the `User` who made the winning bid. (Of course, `anItem.getSuccessfulBid().getBidder()` can provide the same access with a different path.) If made bidirectional, this association will also help to render a screen that shows all auctions a particular user has won: You call `aUser.getBoughtItems()` instead of writing a query.

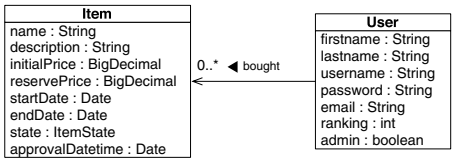

Figure 7.7
Items may be bought by users.

From the point of view of the `User` class, the association is one-to-many. The classes and their relationship are shown in figure 7.7.

Why is this association different than the one between `Item` and `Bid`? The multiplicity `0..*` in UML indicates that the reference is *optional*. This doesn't influence the Java domain model much, but it has consequences for the underlying tables. You expect a `BUYER_ID` foreign key column in the `ITEM` table. The column has to be nullable—a particular `Item` may not have been bought (as long as the auction is still running).

You can accept that the foreign key column can be `NULL` and apply additional constraints ("allowed to be `NULL` only if the auction end time hasn't been reached or if no bid has been made"). We always try to avoid nullable columns in a relational database schema. Information that is unknown degrades the quality of the data you store. Tuples represent propositions that are *true*; you can't assert something you don't know. And, in practice, many developers and DBAs don't create the right constraint and rely on (often buggy) application code to provide data integrity.

An optional entity association, be it one-to-one or one-to-many, is best represented in an SQL database with a join table. See figure 7.8 for an example schema.

You added a join table earlier in this chapter, for a one-to-one association. To guarantee the multiplicity of one-to-one, you applied *unique* constraints on both foreign key columns of the join table. In the current case, you have a one-to-many multiplicity, so only the `ITEM_ID` column of the `ITEM_BUYER` table is unique. A particular item can be bought only once.

Let's map this in XML. First, here's the `boughtItems` collection of the `User` class.

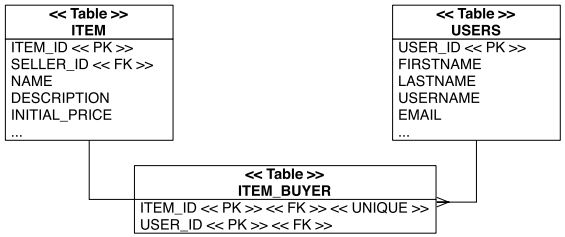

Figure 7.8 An optional relationship with a join table avoids nullable foreign key columns.

```
<set name="boughtItems" table="ITEM_BUYER">
    <key column="USER_ID"/>
    <many-to-many class="Item"
                  column="ITEM_ID"
                  unique="true"/>
</set>
```

You use a `Set` as the collection type. The collection table is the join table, `ITEM_BUYER`; its primary key is a composite of `USER_ID` and `ITEM_ID`. The new mapping element you haven't seen before is `<many-to-many>`; it's required because the regular `<one-to-many>` doesn't know anything about join tables. By forcing a `unique` constraint on the foreign key column that references the target entity table, you effectively force a one-to-many multiplicity.

You can map this association bidirectional with the `buyer` property of `Item`. Without the join table, you'd add a `<many-to-one>` with a `BUYER_ID` foreign key column in the `ITEM` table. With the join table, you have to move this foreign key column into the join table. This is possible with a `<join>` mapping:

```
<join table="ITEM_BUYER"
      optional="true"
      inverse="true">
     <key column="ITEM_ID" unique="true" not-null="true"/>
     <many-to-one name="buyer" column="USER_ID"/>
</join>
```

Two important details: First, the association is optional, and you tell Hibernate not to insert a row into the join table if the grouped properties (only one here, `buyer`) are `null`. Second, this is a bidirectional entity association. As always, one side has to be the inverse end. You've chosen the `<join>` to be inverse; Hibernate now uses the collection state to synchronize the database and ignores the state of

the `Item.buyer` property. As long as your collection is not an indexed variation (a list, map, or array), you can reverse this by declaring the collection `inverse="true"`. The Java code to create a link between a bought item and a user object is the same in both cases:

```
aUser.getBoughtItems().add(anItem);
anItem.setBuyer(aUser);
```

You can map secondary tables in JPA to create a one-to-many association with a join table. First, map a `@ManyToOne` to a join table:

```
@Entity
public class Item {
    @ManyToOne
    @JoinTable(
        name = "ITEM_BUYER",
        joinColumns = {@JoinColumn(name = "ITEM_ID")},
        inverseJoinColumns = {@JoinColumn(name = "USER_ID")}
    )
    private User buyer;
    ...
}
```

At the time of writing, this mapping has the limitation that you can't set it to `optional="true"`; hence, the `USER_ID` column is nullable. If you try to add a `nullable="false"` attribute on the `@JoinColumn`, Hibernate Annotations thinks that you want the whole `buyer` property to never be `null`. Furthermore, the primary key of the join table is now the `ITEM_ID` column only. This is fine, because you don't want duplicate items in this table—they can be bought only once.

To make this mapping bidirectional, add a collection on the `User` class and make it inverse with `mappedBy`:

```
@OneToMany(mappedBy = "buyer")
private Set<Item> boughtItems = new HashSet<Item>();
```

We showed a `<many-to-many>` XML mapping element in the previous section for a one-to-many association on a join table. The `@JoinTable` annotation is the equivalent in annotations. Let's map a real many-to-many association.

7.2.2 Many-to-many associations

The association between `Category` and `Item` is a many-to-many association, as can be seen in figure 7.9.

In a real system, you may not have a many-to-many association. Our experience is that there is almost always other information that must be attached to each link between associated instances (such as the date and time when an item was added

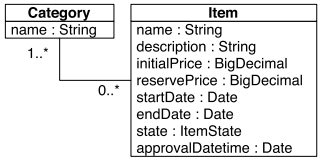

Figure 7.9
A many-to-many valued association between `Category` and `Item`

to a category) and that the best way to represent this information is via an intermediate *association class*. In Hibernate, you can map the association class as an entity and map two one-to-many associations for either side. Perhaps more conveniently, you can also map a composite element class, a technique we show later.
It's the purpose of this section to implement a real many-to-many entity association. Let's start with a unidirectional example.

A simple unidirectional many-to-many association

If you require only unidirectional navigation, the mapping is straightforward. Unidirectional many-to-many associations are essentially no more difficult than the collections of value-type instances we discussed earlier. For example, if the `Category` has a set of `Items`, you can create this mapping:

```
<set name="items"
     table="CATEGORY_ITEM"
     cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
```

The join table (or *link table*, as some developers call it) has two columns: the foreign keys of the `CATEGORY` and `ITEM` tables. The primary key is a composite of both columns. The full table structure is shown in figure 7.10.

In JPA annotations, many-to-many associations are mapped with the `@ManyToMany` attribute:

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```


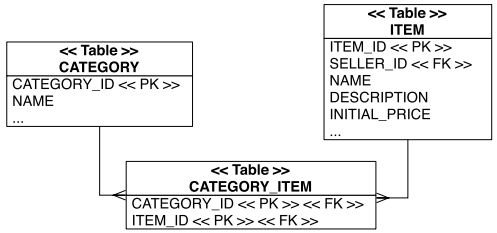

Figure 7.10 Many-to-many entity association mapped to an association table

In Hibernate XML you can also switch to an `<idbag>` with a separate primary key column on the join table:

```
<idbag name="items"
       table="CATEGORY_ITEM"
       cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
```

As usual with an `<idbag>` mapping, the primary key is a surrogate key column, `CATEGORY_ITEM_ID`. Duplicate links are therefore allowed; the same `Item` can be added twice to a `Category`. (This doesn't seem to be a useful feature.) With annotations, you can switch to an identifier bag with the Hibernate `@CollectionId`:

```
@ManyToMany
@CollectionId(
    columns = @Column(name = "CATEGORY_ITEM_ID"),
    type = @org.hibernate.annotations.Type(type = "long"),
    generator = "sequence"
)
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Collection<Item> items = new ArrayList<Item>();
```

A JPA XML descriptor for a regular many-to-many mapping with a set (you can't use a Hibernate extension for identifier bags) looks like this:

```
<entity class="auction.model.Category" access="FIELD">
    ...
    <many-to-many name="items">
        <join-table name="CATEGORY_ITEM">
            <join-column name="CATEGORY_ID"/>
            <inverse-join-column name="ITEM_ID"/>
        </join-table>
    </many-to-many>

</entity>
```

You may even switch to an indexed collection (a map or list) in a many-to-many association. The following example maps a list in Hibernate XML:

```
<list name="items"
      table="CATEGORY_ITEM"
      cascade="save-update">
    <key column="CATEGORY_ID"/>
    <list-index column="DISPLAY_POSITION"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</list>
```

The primary key of the link table is a composite of the `CATEGORY_ID` and `DISPLAY_POSITION` columns; this mapping guarantees that the position of each `Item` in a `Category` is persistent. Or, with annotations:

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
@org.hibernate.annotations.IndexColumn(name = "DISPLAY_POSITION")
private List<Item> items = new ArrayList<Item>();
```

As discussed earlier, JPA only supports ordered collections (with an optional `@OrderBy` annotation or ordered by primary key), so you again have to use a Hibernate extension for indexed collection support. If you don't add an `@IndexColumn`, the `List` is stored with bag semantics (no guaranteed persistent order of elements).

Creating a link between a `Category` and an `Item` is easy:

```
aCategory.getItems().add(anItem);
```

Bidirectional many-to-many associations are slightly more difficult.

A bidirectional many-to-many association

You know that one side in a bidirectional association has to be mapped as inverse because you have named the foreign key column(s) twice. The same principle applies to bidirectional many-to-many associations: Each row of the link table is represented by two collection elements, one element at each end of the association. An association between an `Item` and a `Category` is represented in memory by the `Item` instance in the `items` collection of the `Category`, but also by the `Category` instance in the `categories` collection of the `Item`.

Before we discuss the mapping of this bidirectional case, you have to be aware that the code to create the object association also changes:

```
aCategory.getItems().add(anItem);
anItem.getCategories().add(aCategory);
```

As always, a bidirectional association (no matter of what multiplicity) requires that you set both ends of the association.

When you map a bidirectional many-to-many association, you must declare one end of the association using `inverse="true"` to define which side's state is used to update the join table. You can choose which side should be inverse.

Recall this mapping of the `items` collection from the previous section:

```
<class name="Category" table="CATEGORY">
    ...
    <set name="items"
         table="CATEGORY_ITEM"
         cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
```

You may reuse this mapping for the `Category` end of the bidirectional association and map the other side as follows:

```
<class name="Item" table="ITEM">
    ...
    <set name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

Note the `inverse="true"`. Again, this setting tells Hibernate to ignore changes made to the `categories` collection and that the other end of the association, the `items` collection, is the representation that should be synchronized with the database if you link instances in Java code.

You have enabled `cascade="save-update"` for both ends of the collection. This isn't unreasonable, we suppose. On the other hand, the cascading options `all`, `delete`, and `delete-orphans` aren't meaningful for many-to-many associations. (This is good point to test if you understand entities and value types—try to come up with reasonable answers why these cascading options don't make sense for a many-to-many association.)

In JPA and with annotations, making a many-to-many association bidirectional is easy. First, the noninverse side:

```
@ManyToMany
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = {@JoinColumn(name = "CATEGORY_ID")},
    inverseJoinColumns = {@JoinColumn(name = "ITEM_ID")}
)
private Set<Item> items = new HashSet<Item>();
```

Now the opposite inverse side:

```
@ManyToMany(mappedBy = "items")
private Set<Category> categories = new HashSet<Category>();
```

As you can see, you don't have to repeat the join-table declaration on the inverse side.

What types of collections may be used for bidirectional many-to-many associations? Do you need the same type of collection at each end? It's reasonable to map, for example, a `<list>` for the noninverse side of the association and a `<bag>` on the inverse side.

For the inverse end, `<set>` is acceptable, as is the following bag mapping:

```
<class name="Item" table="ITEM">
    ...
    <bag name="categories"
         table="CATEGORY_ITEM"
         inverse="true"
         cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

In JPA, a bag is a collection without a persistent index:

```
@ManyToMany(mappedBy = "items")
private Collection<Category> categories = new ArrayList<Category>();
```


No other mappings can be used for the inverse end of a many-to-many association. Indexed collections (lists and maps) don't work, because Hibernate won't initialize or maintain the index column if the collection is inverse. In other words, a many-to-many association can't be mapped with indexed collections on both sides.

We already frowned at the use of many-to-many associations, because additional columns on the join table are almost always inevitable.

7.2.3 Adding columns to join tables

In this section, we discuss a question that is asked frequently by Hibernate users: What do I do if my join table has additional columns, not only two foreign key columns?

Imagine that you need to record some information each time you add an `Item` to a `Category`. For example, you may need to store the date and the name of the user who added the item to this category. This requires additional columns on the join table, as you can see in figure 7.11.

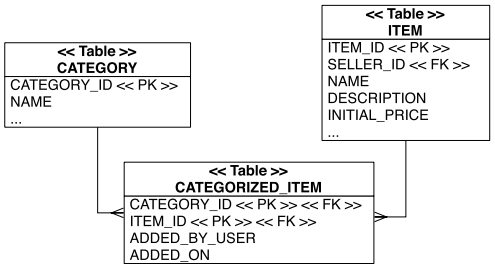

Figure 7.11 Additional columns on the join table in a many-to-many association

You can use two common strategies to map such a structure to Java classes. The first strategy requires an intermediate entity class for the join table and is mapped with one-to-many associations. The second strategy utilizes a collection of components, with a value-type class for the join table.

Mapping the join table to an intermediate entity

The first option we discuss now resolves the many-to-many relationship between `Category` and `Item` with an intermediate entity class, `CategorizedItem`. Listing 7.1 shows this entity class, which represents the join table in Java, including JPA annotations:

Listing 7.1 An entity class that represents a link table with additional columns

```
@Entity
@Table(name = "CATEGORIZED_ITEM")
public class CategorizedItem {

    @Embeddable
    public static class Id implements Serializable {

        @Column(name = "CATEGORY_ID")
        private Long categoryId;

        @Column(name = "ITEM_ID")
        private Long itemId;

        public Id() {}

        public Id(Long categoryId, Long itemId) {
            this.categoryId = categoryId;
            this.itemId = itemId;
        }
        public boolean equals(Object o) {
            if (o != null && o instanceof Id) {
                Id that = (Id)o;
                return this.categoryId.equals(that.categoryId) &&
                       this.itemId.equals(that.itemId);
            } else {
                return false;
            }
        }

        public int hashCode() {
            return categoryId.hashCode() + itemId.hashCode();
        }
    }

    @EmbeddedId
    private Id id = new Id();

    @Column(name = "ADDED_BY_USER")
    private String username;

    @Column(name = "ADDED_ON")
    private Date dateAdded = new Date();

    @ManyToOne
    @JoinColumn(name="ITEM_ID",
                insertable = false,
                updatable = false)
    private Item item;

    @ManyToOne
    @JoinColumn(name="CATEGORY_ID",
                insertable = false,
                updatable = false)
    private Category category;

    public CategorizedItem() {}

    public CategorizedItem(String username,
                           Category category,
                           Item item) {
        // Set fields
        this.username = username;

        this.category = category;
        this.item = item;

        // Set identifier values
        this.id.categoryId = category.getId();
        this.id.itemId = item.getId();

        // Guarantee referential integrity
        category.getCategorizedItems().add(this);
        item.getCategorizedItems().add(this);
    }

    // Getter and setter methods
    ...
}
```


An entity class needs an identifier property. The primary key of the join table is `CATEGORY_ID` and `ITEM_ID`, a composite. Hence, the entity class also has a composite key, which you encapsulate in a static nested class for convenience. You can also see that constructing a `CategorizedItem` involves setting the values of the identifier—composite key values are assigned by the application. Pay extra attention to the constructor and how it sets the field values and guarantees referential integrity by managing collections on either side of the association.

Let's map this class to the join table in XML:

```
<class name="CategorizedItem"
       table="CATEGORY_ITEM"
       mutable="false">

    <composite-id name="id" class="CategorizedItem$Id">
        <key-property name="categoryId"
                      access="field"
                      column="CATEGORY_ID"/>

        <key-property name="itemId"
                      access="field"
                      column="ITEM_ID"/>
    </composite-id>

    <property name="dateAdded"
              column="ADDED_ON"
              type="timestamp"
```

```
            not-null="true"/>

    <property name="username"
              column="ADDED_BY_USER"
              type="string"
              not-null="true"/>

    <many-to-one name="category"
                 column="CATEGORY_ID"
                 not-null="true"
                 insert="false"
                 update="false"/>

    <many-to-one name="item"
                 column="ITEM_ID"
                 not-null="true"
                 insert="false"
                 update="false"/>

</class>
```

The entity class is mapped as immutable—you'll never update any properties after creation. Hibernate accesses `<composite-id>` fields directly—you don't need getters and setters in this nested class. The two `<many-to-one>` mappings are effectively read-only; `insert` and `update` are set to `false`. This is necessary because the columns are mapped twice, once in the composite key (which is responsible for insertion of the values) and again for the many-to-one associations.

The `Category` and `Item` entities (can) have a one-to-many association to the `CategorizedItem` entity, a collection. For example, in `Category`:

```
<set name="categorizedItems"
     inverse="true">
    <key column="CATEGORY_ID"/>
    <one-to-many class="CategorizedItem"/>
</set>
```

And here's the annotation equivalent:

```
@OneToMany(mappedBy = "category")
private Set<CategorizedItem> categorizedItems =
                                new HashSet<CategorizedItem>();
```

There is nothing special to consider here; it's a regular bidirectional one-to-many association with an inverse collection. Add the same collection and mapping to `Item` to complete the association. This code creates and stores a link between a category and an item:

```
CategorizedItem newLink =
    new CategorizedItem(aUser.getUsername(), aCategory, anItem);

session.save(newLink);
```


The referential integrity of the Java objects is guaranteed by the constructor of `CategorizedItem`, which manages the collection in `aCategory` and in `anItem`. Remove and delete the link between a category and an item:

```
aCategory.getCategorizedItems().remove( theLink );
anItem.getCategorizedItems().remove( theLink );

session.delete(theLink);
```

The primary advantage of this strategy is the possibility for bidirectional navigation: You can get all items in a category by calling `aCategory.getCategorizedItems()` and the also navigate from the opposite direction with `anItem.getCategorizedItems()`. A disadvantage is the more complex code needed to manage the `CategorizedItem` entity instances to create and remove associations—they have to be saved and deleted independently, and you need some infrastructure in the `CategorizedItem` class, such as the composite identifier. However, you can enable transitive persistence with cascading options on the collections from `Category` and `Item` to `CategorizedItem`, as explained in chapter 12, section 12.1, "Transitive persistence."

The second strategy for dealing with additional columns on the join table doesn't need an intermediate entity class; it's simpler.

Mapping the join table to a collection of components

First, simplify the `CategorizedItem` class, and make it a value type, without an identifier or any complex constructor:

```
public class CategorizedItem {
    private String username;
    private Date dateAdded = new Date();
    private Item item;
    private Category category;

    public CategorizedItem(String username,
                           Category category,
                           Item item) {
        this.username = username;
        this.category = category;
        this.item = item;
    }
    ...

    // Getter and setter methods
    // Don't forget the equals/hashCode methods
}
```

As for all value types, this class has to be owned by an entity. The owner is the `Category`, and it has a collection of these components:

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

    ...
    <set name="categorizedItems" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <composite-element class="CategorizedItem">
            <parent name="category"/>

            <many-to-one name="item"
                         column="ITEM_ID"
                         not-null="true"
                         class="Item"/>

            <property name="username" column="ADDED_BY_USER"/>
            <property name="dateAdded" column="ADDED_ON"/>

        </composite-element>
     </set>

</class>
```

This is the complete mapping for a many-to-many association with extra columns on the join table. The `<many-to-one>` element represents the association to `Item`; the `<property>` mappings cover the extra columns on the join table. There is only one change to the database tables: The `CATEGORY_ITEM` table now has a primary key that is a composite of all columns, not only `CATEGORY_ID` and `ITEM_ID`, as in the previous section. Hence, all properties should never be `nullable`—otherwise you can't identify a row in the join table. Except for this change, the tables still look as shown in figure 7.11.

You can enhance this mapping with a reference to the `User` instead of just the user's name. This requires an additional `USER_ID` column on the join table, with a foreign key to `USERS`. This is a *ternary association* mapping:

```
<set name="categorizedItems" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <composite-element class="CategorizedItem">
        <parent name="category"/>

        <many-to-one name="item"
                     column="ITEM_ID"
                     not-null="true"
                     class="Item"/>

        <many-to-one name="user"
                     column="USER_ID"
                     not-null="true"
                     class="User"/>

        <property name="dateAdded" column="ADDED_ON"/>

    </composite-element>
</set>
```


This is a fairly exotic beast!

The advantage of a collection of components is clearly the implicit lifecycle of the link objects. To create an association between a `Category` and an `Item`, add a new `CategorizedItem` instance to the collection. To break the link, remove the element from the collection. No extra cascading settings are required, and the Java code is simplified:

```
CategorizedItem aLink =
    new CategorizedItem(aUser.getUserName(), aCategory, anItem);

aCategory.getCategorizedItems().add( aLink );

aCategory.getCategorizedItems().remove( aLink );
```

The downside of this approach is that there is no way to enable bidirectional navigation: A component (such as `CategorizedItem`) can't, by definition, have shared references. You can't navigate from `Item` to `CategorizedItem`. However, you can write a query to retrieve the objects you need.

Let's do the same mapping with annotations. First, make the component class `@Embeddable`, and add the component column and association mappings:

```
@Embeddable
public class CategorizedItem {

    @org.hibernate.annotations.Parent // Optional back-pointer
    private Category category;

    @ManyToOne
    @JoinColumn(name = "ITEM_ID",
                nullable = false,
                updatable = false)
    private Item item;

    @ManyToOne
    @JoinColumn(name = "USER_ID",
                nullable = false,
                updatable = false)
    private User user;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "ADDED_ON", nullable = false, updatable = false)
    private Date dateAdded;

    ...
    // Constructor
    // Getter and setter methods
    // Don't forget the equals/hashCode methods
}
```

Now map this as a collection of components in the `Category` class:

```
@org.hibernate.annotations.CollectionOfElements
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = @JoinColumn(name = "CATEGORY_ID")
)
private Set<CategorizedItem> categorizedItems =
                                new HashSet<CategorizedItem>();
```

That's it: You've mapped a ternary association with annotations. What looked incredibly complex at the beginning has been reduced to a few lines of annotation metadata, most of it optional.

The last collection mapping we'll explore are `Map`s of entity references.

7.2.4 Mapping maps

You mapped a Java `Map` in the last chapter—the keys and values of the `Map` were value types, simple strings. You can create more complex maps; not only can the keys be references to entities, but so can the values. The result can therefore be a ternary association.

Values as references to entities

First, let's assume that only the value of each map entry is a reference to another entity. The key is a value type, a `long`. Imagine that the `Item` entity has a map of `Bid` instances and that each map entry is a pair of `Bid` identifier and reference to a `Bid` instance. If you iterate through `anItem.getBidsByIdentifier()`, you iterate through map entries that look like `(1, <reference to Bid with PK 1>)`, `(2, <reference to Bid with PK 2>)`, and so on.

The underlying tables for this mapping are nothing special; you again have an `ITEM` and a `BID` table, with an `ITEM_ID` foreign key column in the `BID` table. Your motivation here is a slightly different representation of the data in the application, with a `Map`.

In the `Item` class, include a `Map`:

```
@MapKey(name="id")
@OneToMany
private Map<Long,Bid> bidsByIdentifier = new HashMap<Long,Bid>();
```

New here is the `@MapKey` element of JPA—it maps a property of the target entity as key of the map.The default if you omit the `name` attribute is the identifier property of the target entity (so the name here is redundant). Because the keys of a map form a set, values are expected to be unique for a particular map—this is the case for `Bid` primary keys but likely not for any other property of `Bid`.

In Hibernate XML, this mapping is as follows:

```
<map name="bidsByIdentifier">
    <key column="ITEM_ID"/>
    <map-key type="long" formula="BID_ID"/>
    <one-to-many class="Bid"/>
</map>
```

The `formula` key for a map makes this column read-only, so it's never updated when you modify the map. A more common situation is a map in the middle of a ternary association.

Ternary associations

You may be a little bored by now, but we promise this is the last time we'll show another way to map the association between `Category` and `Item`. Let's summarize what you already know about this many-to-many association:

- It can be mapped with two collections on either side and a join table that has only two foreign key columns. This is a regular many-to-many association mapping.
- It can be mapped with an intermediate entity class that represents the join table, and any additional columns therein. A one-to-many association is mapped on either side (`Category` and `Item`), and a bidirectional many-to-one equivalent is mapped in the intermediate entity class.
- It can be mapped unidirectional, with a join table represented as a value type component. The `Category` entity has a collection of components. Each component has a reference to its owning `Category` and a many-to-one entity association to an `Item`. (You can also switch the words `Category` and `Item` in this explanation.)

You previously turned the last scenario into a ternary association by adding another many-to-one entity association to a `User`. Let's do the same with a `Map`.

A `Category` has a `Map` of `Item` instances—the key of each map entry is a reference to an `Item`. The value of each map entry is the `User` who added the `Item` to the `Category`. This strategy is appropriate if there are no additional columns on the join table; see the schema in figure 7.12.

The advantage of this strategy is that you don't need any intermediate class, no entity or value type, to represent the `ADDED_BY_USER_ID` column of the join table in your Java application.

First, here's the `Map` property in `Category` with a Hibernate extension annotation.

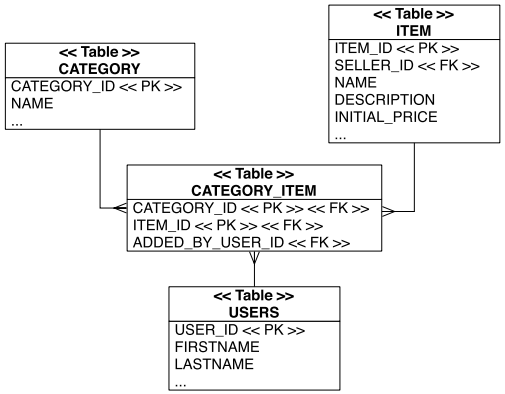

Figure 7.12 A ternary association with a join table between three entities

```
@ManyToMany
@org.hibernate.annotations.MapKeyManyToMany(
    joinColumns = @JoinColumn(name = "ITEM_ID")
)
@JoinTable(
    name = "CATEGORY_ITEM",
    joinColumns = @JoinColumn(name = "CATEGORY_ID"),
    inverseJoinColumns = @JoinColumn(name = "USER_ID")
)
private Map<Item,User> itemsAndUser = new HashMap<Item,User>();
```

The Hibernate XML mapping includes a new element, `<map-key-many-to-many>`:

```
<map name="itemsAndUser" table="CATEGORY_ITEM">
    <key column="CATEGORY_ID"/>
    <map-key-many-to-many column="ITEM_ID" class="Item"/>
    <many-to-many column="ADDED_BY_USER_ID" class="User"/>
</map>
```

To create a link between all three entities, if all your instances are already in persistent state, add a new entry to the map:

```
aCategory.getItemsAndUser().add( anItem, aUser );
```

To remove the link, remove the entry from the map. As an exercise, you can try to make this mapping bidirectional, with a collection of `categories` in `Item`.

Remember that this has to be an inverse collection mapping, so it doesn't support indexed collections.

Now that you know all the association mapping techniques for normal entities, we still have to consider inheritance and associations to the various levels of an inheritance hierarchy. What we really want is *polymorphic* behavior. Let's see how Hibernate deals with polymorphic entity associations.

7.3 Polymorphic associations

Polymorphism is a defining feature of object-oriented languages like Java. Support for polymorphic associations and polymorphic queries is an absolutely basic feature of an ORM solution like Hibernate. Surprisingly, we've managed to get this far without needing to talk much about polymorphism. Even more surprisingly, there is not much to say on the topic—polymorphism is so easy to use in Hibernate that we don't need to spend a lot of effort explaining it.

To get an overview, we first consider a many-to-one association to a class that may have subclasses. In this case, Hibernate guarantees that you can create links to any subclass instance just like you would to instances of the superclass.

7.3.1 Polymorphic many-to-one associations

A *polymorphic association* is an association that may refer instances of a subclass of the class that was explicitly specified in the mapping metadata. For this example, consider the `defaultBillingDetails` property of `User`. It references one particular `BillingDetails` object, which at runtime can be any concrete instance of that class. The classes are shown in figure 7.13.

You map this association to the abstract class `BillingDetails` as follows in User.hbm.xml.

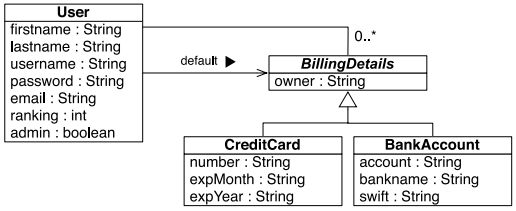

Figure 7.13 A user has either a credit card or a bank account as the default.

```
<many-to-one name="defaultBillingDetails"
             class="BillingDetails"
             column="DEFAULT_BILLING_DETAILS_ID"/>
```

But because `BillingDetails` is abstract, the association must refer to an instance of one of its subclasses—`CreditCard` or `CheckingAccount`—at runtime.

You don't have to do anything special to enable polymorphic associations in Hibernate; specify the name of any mapped persistent class in your association mapping (or let Hibernate discover it using reflection), and then, if that class declares any `<union-subclass>`, `<subclass>`, or `<joined-subclass>` elements, the association is naturally polymorphic.

The following code demonstrates the creation of an association to an instance of the `CreditCard` subclass:

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

User user = (User) session.get(User.class, userId);
user.addBillingDetails(cc); // Add it to the one-to-many association

user.setDefaultBillingDetails(cc);

// Complete unit of work
```

Now, when you navigate the association in a second unit of work, Hibernate automatically retrieves the `CreditCard` instance:

```
User user = (User) secondSession.get(User.class, userId);

// Invoke the pay() method on the actual subclass instance
user.getDefaultBillingDetails().pay(amount);
```

There is just one thing to watch out for: If `BillingDetails` was mapped with `lazy="true"` (which is the default), Hibernate would proxy the `defaultBillingDetails` association target. In this case, you wouldn't be able to perform a typecast to the concrete class `CreditCard` at runtime, and even the `instanceof` operator would behave strangely:

```
User user = (User) session.get(User.class, userid);
BillingDetails bd = user.getDefaultBillingDetails();
System.out.println( bd instanceof CreditCard ); // Prints "false"
CreditCard cc = (CreditCard) bd; // ClassCastException!
```

In this code, the typecast fails because `bd` is a proxy instance. When a method is invoked on the proxy, the call is delegated to an instance of `CreditCard` that is fetched lazily (it's an instance of a runtime-generated subclass, so `instanceof` also fails). Until this initialization occurs, Hibernate doesn't know what the subtype of the given instance is—this would require a database hit, which you try to avoid with lazy loading in the first place. To perform a proxy-safe typecast, use `load()`:


```
User user = (User) session.get(User.class, userId);
BillingDetails bd = user.getDefaultBillingDetails();

// Narrow the proxy to the subclass, doesn't hit the database
CreditCard cc =
    (CreditCard) session.load( CreditCard.class, bd.getId() );
expiryDate = cc.getExpiryDate();
```

After the call to `load()`, `bd` and `cc` refer to two different proxy instances, which both delegate to the same underlying `CreditCard` instance. However, the second proxy has a different interface, and you can call methods (like `getExpiryDate()`) that apply only to this interface.

Note that you can avoid these issues by avoiding lazy fetching, as in the following code, using an eager fetch query:

```
User user = (User)session.createCriteria(User.class)
    .add(Restrictions.eq("id", uid) )
    .setFetchMode("defaultBillingDetails", FetchMode.JOIN)
    .uniqueResult();

// The users defaultBillingDetails have been fetched eagerly
CreditCard cc = (CreditCard) user.getDefaultBillingDetails();
expiryDate = cc.getExpiryDate();
```

Truly object-oriented code shouldn't use `instanceof` or numerous typecasts. If you find yourself running into problems with proxies, you should question your design, asking whether there is a more polymorphic approach. Hibernate also offers bytecode instrumentation as an alternative to lazy loading through proxies; we'll get back to fetching strategies in chapter 13, section 13.1, "Defining the global fetch plan."

One-to-one associations are handled the same way. What about many-valued associations—for example, the collection of `billingDetails` for each `User`?

7.3.2 Polymorphic collections

A `User` may have references to many `BillingDetails`, not only a single default (one of the many *is* the default). You map this with a bidirectional one-to-many association.

In `BillingDetails`, you have the following:

```
<many-to-one name="user"
             class="User"
             column="USER_ID"/>
```

In the `User`s mapping you have:

```
<set name="billingDetails"
     inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

Adding a `CreditCard` is easy:

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpMonth(...);
cc.setExpYear(...);

User user = (User) session.get(User.class, userId);

// Call convenience method that sets both sides of the association
user.addBillingDetails(cc);

// Complete unit of work
```

As usual, `addBillingDetails()` calls `getBillingDetails().add(cc)` and `cc.setUser(this)` to guarantee the integrity of the relationship by setting both pointers.

You may iterate over the collection and handle instances of `CreditCard` and `CheckingAccount` polymorphically (you probably don't want to bill users several times in the final system, though):

```
User user = (User) session.get(User.class, userId);

for( BillingDetails bd : user.getBillingDetails() ) {
    // Invoke CreditCard.pay() or BankAccount.pay()
    bd.pay(paymentAmount);
}
```

In the examples so far, we assumed that `BillingDetails` is a class mapped explicitly and that the inheritance mapping strategy is *table per class hierarchy*, or normalized with *table per subclass*.

However, if the hierarchy is mapped with *table per concrete class* (implicit polymorphism) or explicitly with *table per concrete class with union*, this scenario requires a more sophisticated solution.

7.3.3 Polymorphic associations to unions

Hibernate supports the polymorphic many-to-one and one-to-many associations shown in the previous sections even if a class hierarchy is mapped with the *table per concrete class* strategy. You may wonder how this works, because you may not have a table for the superclass with this strategy; if so, you can't reference or add a foreign key column to `BILLING_DETAILS`.

Review our discussion of *table per concrete class with union* in chapter 5, section 5.1.2, "Table per concrete class with unions." Pay extra attention to the polymorphic query Hibernate executes when retrieving instances of `BillingDetails`. Now, consider the following collection of `BillingDetails` mapped for `User`:

```
<set name="billingDetails"
     inverse="true">
    <key column="USER_ID"/>
    <one-to-many class="BillingDetails"/>
</set>
```

If you want to enable the polymorphic union feature, a requirement for this polymorphic association is that it's inverse; there must be a mapping on the opposite side. In the mapping of `BillingDetails`, with `<union-subclass>`, you have to include a `<many-to-one>` association:

```
<class name="BillingDetails" abstract="true">

    <id name="id" column="BILLING_DETAILS_ID" .../>

    <property .../>

    <many-to-one name="user"
                 column="USER_ID"
                 class="User"/>

    <union-subclass name="CreditCard" table="CREDIT_CARD">
        <property .../>
    </union-subclass>

    <union-subclass name="BankAccount" table="BANK_ACCOUNT">
        <property .../>
    </union-subclass>

</class>
```

You have two tables for both concrete classes of the hierarchy. Each table has a foreign key column, `USER_ID`, referencing the `USERS` table. The schema is shown in figure 7.14.

Now, consider the following data-access code:

```
aUser.getBillingDetails().iterator().next();
```

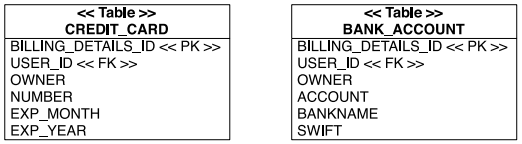

Figure 7.14 Two concrete classes mapped to two separate tables

Hibernate executes a UNION query to retrieve all instances that are referenced in this collection:

```
select
    BD.*
from
  ( select
      BILLING_DETAILS_ID, USER_ID, OWNER,
      NUMBER, EXP_MONTH, EXP_YEAR,
      null as ACCOUNT, null as BANKNAME, null as SWIFT,
      1 as CLAZZ
    from
      CREDIT_CARD

    union

    select
      BILLING_DETAILS_ID, USER_ID, OWNER,
      null as NUMBER, null as EXP_MONTH, null as EXP_YEAR
      ACCOUNT, BANKNAME, SWIFT,
      2 as CLAZZ
    from
      BANK_ACCOUNT
   ) BD
where
    BD.USER_ID = ?
```

The FROM-clause subselect is a union of all concrete class tables, and it includes the USER_ID foreign key values for all instances. The outer select now includes a restriction in the WHERE clause to all rows referencing a particular user.

This magic works great for retrieval of data. If you manipulate the collection and association, the noninverse side is used to update the USER_ID column(s) in the concrete table. In other words, the modification of the inverse collection has no effect: The value of the user property of a CreditCard or BankAccount instance is taken.

Now consider the many-to-one association defaultBillingDetails again, mapped with the DEFAULT_BILLING_DETAILS_ID column in the USERS table. Hibernate executes a UNION query that looks similar to the previous query to retrieve this instance, if you access the property. However, instead of a restriction in the WHERE clause to a particular user, the restriction is made on a particular BILLING_DETAILS_ID.

Important: Hibernate cannot and will not create a foreign key constraint for DEFAULT_BILLING_DETAILS_ID with this strategy. The target table of this reference can be any of the concrete tables, which can't be constrained easily. You should consider writing a custom integrity rule for this column with a database trigger.

One problematic inheritance strategy remains: *table per concrete class* with implicit polymorphism.

7.3.4 Polymorphic table per concrete class

In chapter 5, section 5.1.1, "Table per concrete class with implicit polymorphism," we defined the *table per concrete class* mapping strategy and observed that this mapping strategy makes it difficult to represent a polymorphic association, because you can't map a foreign key relationship to a table of the abstract superclass. There is no table for the superclass with this strategy; you have tables only for concrete classes. You also can't create a UNION, because Hibernate doesn't know what unifies the concrete classes; the superclass (or interface) isn't mapped anywhere.

Hibernate doesn't support a polymorphic billingDetails one-to-many collection in User, if this inheritance mapping strategy is applied on the BillingDetails hierarchy. If you need polymorphic many-to-one associations with this strategy, you'll have to resort to a hack. The technique we'll show you in this section should be your last choice. Try to switch to a <union-subclass> mapping first.

Suppose that you want to represent a polymorphic many-to-one association from User to BillingDetails, where the BillingDetails class hierarchy is mapped with a *table per concrete class* strategy and implicit polymorphic behavior in Hibernate. You have a CREDIT_CARD table and a BANK_ACCOUNT table, but no BILLING_DETAILS table. Hibernate needs two pieces of information in the USERS table to uniquely identify the associated default CreditCard or BankAccount:

- The name of the table in which the associated instance resides
- The identifier of the associated instance

The USERS table requires a DEFAULT_BILLING_DETAILS_TYPE column in addition to the DEFAULT_BILLING_DETAILS_ID. This extra column works as an additional discriminator and requires a Hibernate <any> mapping in User.hbm.xml:

```
<any name="defaultBillingDetails"
     id-type="long"
     meta-type="string">
    <meta-value value="CREDIT_CARD" class="CreditCard"/>
    <meta-value value="BANK_ACCOUNT" class="BankAccount"/>
    <column name="DEFAULT_BILLING_DETAILS_TYPE"/>
    <column name="DEFAULT_BILLING_DETAILS_ID"/>
</any>
```

The meta-type attribute specifies the Hibernate type of the DEFAULT_BILLING_DETAILS_TYPE column; the id-type attribute specifies the type of the DEFAULT_

BILLING_DETAILS_ID column (it's necessary for CreditCard and BankAccount to have the same identifier type).

The <meta-value> elements tell Hibernate how to interpret the value of the DEFAULT_BILLING_DETAILS_TYPE column. You don't need to use the full table name here—you can use any value you like as a type discriminator. For example, you can encode the information in two characters:

```
<any name="defaultBillingDetails"
     id-type="long"
     meta-type="string">
    <meta-value value="CC" class="CreditCard"/>
    <meta-value value="CA" class="BankAccount"/>
    <column name="DEFAULT_BILLING_DETAILS_TYPE"/>
    <column name="DEFAULT_BILLING_DETAILS_ID"/>
</any>
```

An example of this table structure is shown in figure 7.15.

Here is the first major problem with this kind of association: You can't add a foreign key constraint to the DEFAULT_BILLING_DETAILS_ID column, because some values refer to the BANK_ACCOUNT table and others to the CREDIT_CARD table. Thus, you need to come up with some other way to ensure integrity (a trigger, for example). This is the same issue you'd face with a <union-subclass> strategy.

Furthermore, it's difficult to write SQL table joins for this association. In particular, the Hibernate query facilities don't support this kind of association mapping, nor may this association be fetched using an outer join. We discourage the use of <any> associations for all but the *most* special cases. Also note that this mapping

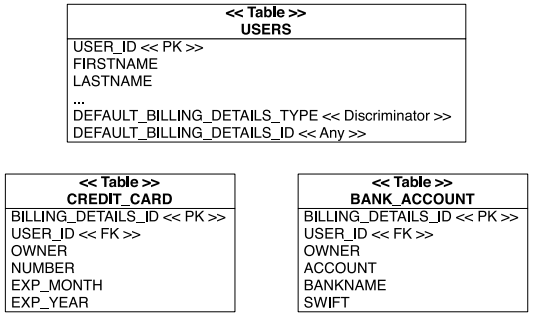

Figure 7.15 Using a discriminator column with an *any* association

technique isn't available with annotations or in Java Persistence (this mapping is so rare that nobody asked for annotation support so far).

As you can see, as long as you don't plan to create an association to a class hierarchy mapped with implicit polymorphism, associations are straightforward; you don't usually need to think about it. You may be surprised that we didn't show any JPA or annotation example in the previous sections—the runtime behavior is the same, and you don't need any extra mapping to get it.

7.4 Summary

In this chapter, you learned how to map more complex entity associations. Many of the techniques we've shown are rarely needed and may be unnecessary if you can simplify the relationships between your classes. In particular, many-to-many entity associations are often best represented as two one-to-many associations to an intermediate entity class, or with a collection of components.

Table 7.1 shows a summary you can use to compare native Hibernate features and Java Persistence.

Table 7.1 Hibernate and JPA comparison chart for chapter 7

Hibernate Core	Java Persistence and EJB 3.0
Hibernate supports key generation for shared primary key one-to-one association mappings.	Standardized one-to-one mapping is supported. Automatic shared primary key generation is possible through a Hibernate extension.
Hibernate supports all entity association mappings across join tables.	Standardized association mappings are available across secondary tables.
Hibernate supports mapping of lists with persistent indexes.	Persistent indexes require a Hibernate extension annotation.
Hibernate supports fully polymorphic behavior. It provides extra support for *any* association mappings to an inheritance hierarchy mapped with implicit polymorphism.	Fully polymorphic behavior is available, but there is no annotation support for *any* mappings.

In the next chapter, we'll focus on legacy database integration and how you can customize the SQL that Hibernate generates automatically for you. This chapter is interesting not only if you have to work with legacy schemas, but also if you want to improve your new schema with custom DDL, for example.