CHAPTER 7

Creating Mappings with Hibernate XML Files

In the simple example programs in Chapters 1 and 3, we demonstrated how a mapping file could be used to establish the relationship between the object model and the database schema. A mapping file can map a single class or multiple classes to the database. The mapping can also describe standard queries (in HQL and SQL) and filters.

Hibernate Types

Although we have referred to the Hibernate types in passing, we have not discussed the terminology in any depth. In order to express the behavior of the mapping file elements, we need to make these fine distinctions explicit.

Hibernate types fall into three broad categories: entities, components, and values.

Entities

Generally, an entity is a POJO class that has been mapped into the database using the `<class>` or `<subclass>` elements.

An entity can also be a dynamic map (actually a `Map` of `Maps`). These are mapped against the database in the same way as a POJO, but with the default entity mode of the `SessionFactory` set to `dynamic-map`.

The advantage of POJOs over the `dynamic-map` approach is that compile-time type safety is retained. Conversely, dynamic maps are a quick way to get up and running when building prototypes.

It is also possible to represent your entities as Dom4J `Document` objects. This is a useful feature when importing and exporting data from a preexisting Hibernate database, but it is not really central to the everyday use of Hibernate.

We recommend that you use the standard entity mode unless you need to sacrifice accuracy for timeliness, so the alternate approaches are not discussed in this chapter—however, we give some simple examples of the Dom4J- and `Map`-based mappings in Appendix A.

Components

Lying somewhere between entities and values are component types. When the class representation is simple and its instances have a strong one-to-one relationship with instances of another class, then it is a good candidate to become a component of that other class.

The component will normally be mapped as columns in the same table that represents most of the other attributes of the owning class, so the strength of the relationship must justify this inclusion. In the following code, the `MacAddress` class might a good candidate for a component relationship.

```
public class NetworkInterface {
   public int id;
   public String name;
   public String manufacturer;
   public MacAddress physicalAddress;
}
```

The advantage of this approach is that it allows you to dispense with the primary key of the component and the join to its containing table. If a poor choice of component is made (for example, when a many-to-one relationship actually holds), then data will be duplicated unnecessarily in the component columns.

Values

Everything that is not an entity or a component is a value. Generally, these correspond to the data types supported by your database, the collection types, and, optionally, some user-defined types.

The details of these mappings will be vendor-specific, so Hibernate provides its own value type names; the Java types are defined in terms of these (see Table 7-1).

Table 7-1. *The Standard Hibernate 3 Value Names*

Hibernate 3 Type	Corresponding Java Type
Primitives and Wrappers	
integer	int, java.lang.Integer
long	long, java.lang.Long
short	short, java.lang.Short
float	float, java.lang.Float
double	double, java.lang.Double
character	char, java.lang.Character
byte	byte, java.lang.Byte
boolean, yes_no, true_false	boolean, java.lang.Boolean
Other Classes	
string	java.lang.String
date, time, timestamp	java.util.Date
calendar, calendar_date	java.util.Calendar

Hibernate 3 Type	Corresponding Java Type
big_decimal	java.math.BigDecimal
big_integer	java.math.BigInteger
locale	java.util.Locale
timezone	java.util.TimeZone
currency	java.util.Currency
class	java.lang.Class
binary	byte[]
text	java.lang.String
serializable	java.io.Serializable
clob	java.sql.Clob
blob	java.sql.Blob

In addition to these standard types, you can create your own. Your user type class should implement either the `org.hibernate.usertype.UserType` interface or the `org.hibernate.usertype.CompositeUserType` interface. Once implemented, a custom type can behave identically to the standard types; though depending on your requirements, it may be necessary to specify multiple column names to contain its values, or to provide initialization parameters for your implementation.

For one-off cases, we recommend that you use components—these have similar behavior, but they can be "created" in the mapping file without needing to write Hibernate-specific code. Unless you propose to make substantial use of a custom type throughout your application, it will not be worth the effort. We do not discuss this feature further in this book.

The Anatomy of a Mapping File

A mapping file is a normal XML file. It is validated against a DTD, which can be downloaded from `http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd`. You can also look through the annotated version at `http://hibernatebook.com`.

The terminology used in the naming of elements and attributes is somewhat confusing at first because it is the point of contact between the jargon of the object-oriented and relational worlds.

The <hibernate-mapping> Element

The root element of any mapping file is `<hibernate-mapping>`. As the top-level element, its attributes mostly define default behaviors and settings to apply to the child elements (see Table 7-2).

Table 7-2. *The* `<hibernate-mapping>` *Attributes*

Attribute	Values	Default	Description
`auto-import`	`true`, `false`	`true`	By default, allows you to use the unqualified class names in Hibernate queries. You would normally only set this to `false` if the class name would otherwise be ambiguous.
`catalog`			The database catalog against which queries should apply.
`default-access`		`property`	The default access type. If set to `property`, then get and set methods are used to access the data. If set to `field`, then the data is accessed directly. Alternatively, you can provide the class name of a `PropertyAccessor` implementation defining any other access mechanism.
`default-cascade`			Defines how (and whether) direct changes to data should affect dependent data by default.
`default-lazy`	`true`, `false`	`true`	Defines whether lazy instantiation is used by default. Generally, the performance benefits are such that you will want to use lazy instantiation whenever possible.
`package`			The package from which all implicit imports are considered to occur.
`schema`			The database schema against which queries should apply.

The default cascade modes available for the `default-cascade` attribute (and for the cascade attributes in all other elements) are as follows:

```
create, merge, delete, save-update, evict, replicate, lock, refresh
```

These correspond to the various possible changes in the lifestyle of the parent object. When set (you can include combinations of them as comma-separated values), the relevant changes to the parent will be cascaded to the relation. For example, you may want to apply the `save-update` cascade option to a class that includes `Set` attributes, so that when new persistent classes are added to these, they will not have to be saved explicitly in the session.

There are also three special options:

```
all, delete-orphan, none
```

`all` specifies that all changes to the parent should be propagated to the relation, and `none` specifies that none should. `delete-orphan` applies only to one-to-many associations, and specifies that the relation should be deleted when it is no longer referenced by the parent.

The required order and cardinality of the child elements of `<hibernate-mapping>` are as follows:

```
(meta*,
 typedef*,
 import*,
 (class | subclass | joined-subclass | union-subclass)*,
 (query | sql-query)*,
 filter-def*)
```

THE ORDER AND CARDINALITY INFORMATION FROM THE DTD

The mapping files used by Hibernate have a great many elements and are somewhat self-referential. For example, the `<component>` element permits you to include within it further `<component>` elements, and within those further `<component>` elements—and so on, ad infinitum.

While we do not quote exhaustively from the mapping file's DTD, we sometimes quote the part of it that specifies the permitted ordering and cardinality (number of occurrences) of the child elements of a given element.

The cardinality is expressed by a symbol after the end of the name of the element: * means "zero or more occurrences," ? means "zero or one occurrences," and no trailing symbol means "exactly one occurrence."

The elements can be grouped using brackets, and where the elements are interchangeable, | (the pipe symbol) means "or."

In practical terms, this allows us to tell from the order and cardinality information quoted for the `hibernate-mapping` file that all of the elements immediately below it are, in fact, optional. We can also see that there is no limit to the number of `<class>` elements that can be included.

You can look up this ordering and cardinality information in the DTD for the mapping file for all the elements, including the ones that we have omitted from this chapter. You will also find within the DTD the specification of which attributes are permitted to each element, the values they may take (when they are constrained), and their default values when provided. We recommend that you look at the DTD for enlightenment whenever you are trying to work out whether a specific mapping file should be syntactically valid.

Throughout this book, we have assumed that the mappings are defined in one mapping file for each significant class that is to be mapped to the database. We suggest that you follow this practice in general, but there are some exceptions to this rule. You may, for instance, find it useful to place `query` and `sql-query` entries into an independent mapping file, particularly when they do not fall clearly into the context of a single class.

The <class> Element

The child element that you will use most often—indeed, in nearly all of your mapping files—is `<class>`. As you have seen in earlier chapters, we generally describe the relationships between Java objects and database entities in the body of the `<class>` element. The `<class>` element permits the following attributes to be defined (see Table 7-3).

Table 7-3. *The* `<class>` *Attributes*

Attribute	Values	Default	Description
`abstract`	`true`, `false`	`false`	The flag that should be set if the class being mapped is abstract.
`batch-size`		`1`	Specifies the number of items that can be batched together when retrieving instances of the class by identifier.
`catalog`			The database catalog against which the queries should apply.

Continued

Table 7-3. *Continued*

Attribute	Values	Default	Description
check			Defines an additional row-level check constraint, effectively adding this as a SQL CHECK(...) clause during table generation (for example, check="salary < 1000000").
discriminator-value			A value used to distinguish between otherwise identical subclasses of a common type persisted to the same table. is null and is not null are permissible values. To distinguish between a Cat and a Dog derivative of the Mammal abstract class, for example, you might use discriminator values of C and D, respectively.
dynamic-insert	true, false	false	Indicates whether all columns should appear in INSERT statements. If the attribute is set to true, null columns will not appear in generated INSERT commands. On very wide tables, this may improve performance; but because insert statements are cached, dynamic-insert can easily produce a performance hit.
dynamic-update	true, false	false	Indicates whether all columns should appear in UPDATE statements. If the attribute is set to true, unchanged columns will not appear in generated UPDATE commands. As with dynamic-insert, this can be tweaked for performance reasons. You must enable dynamic-update if you want to use version-based optimistic locking (discussed in Appendix A).
entity-name			The name of the entity to use in place of the class name (therefore required if dynamic mapping is used).
lazy	true, false		Used to disable or enable lazy fetching against the enclosing mapping's default.
mutable	true, false	true	Used to flag that a class is immutable (allowing Hibernate to make some performance optimizations when dealing with these classes).
name			The fully qualified Java name, or optionally unqualified if the <hibernate-mapping> element declares a package attribute, of the class (or interface) that is to be made persistent.
node			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features.

Attribute	Values	Default	Description
optimistic-lock	none, version	version	Specifies the optimistic locking dirty, all strategy to use. The strategy applies at a class level, but in Hibernate 3 can also be specified (or overridden) at an attribute level. Optimistic locking is discussed in Appendix A.
persister			Allows a custom ClassPersister object to be used when persisting the entity.
polymorphism	implicit, explicit	implicit	Determines how polymorphism is to be used. The default implicit behavior will return instances of the class if superclasses or implemented interfaces are named in the query, and will return subclasses if the class itself is named in the query.
proxy			Specifies a class or an interface to use as the proxy for lazy initialization. Hibernate uses runtime-generated proxies by default, but you can specify your own implementation of org.hibernate.HibernateProxy in their place.
rowid			Flags that row IDs should be used (a database-implementation detail allowing Hibernate to optimize updates).
schema			Optionally overrides the schema specified by the <hibernate-mapping> element.
select-before-update	true, false	false	Flags that Hibernate should carry out extra work to avoid issuing unnecessary UPDATE statements. If set to true, Hibernate issues a SELECT statement before attempting to issue an UPDATE statement in order to ensure that the UPDATE statement is actually required (i.e., that columns have been modified). While this is likely to be less efficient, it can prevent database triggers from being invoked unnecessarily.
subselect			A subselection of the contents of the underlying table. A class can only use a subselect if it is immutable and read-only (because the SQL defined here cannot be reversed). Generally, the use of a database view is preferable.
table			The table name associated with the class (if unspecified, the unqualified class name will be used).
where			An arbitrary SQL where condition to be used when retrieving objects of this class from the table.

Many of these attributes in the <class> element are designed to support preexisting database schemas. In practice, the name attribute is very often the only one set.

The required order and cardinality of the child elements of <class> are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 comment?,
 tuplizer*,
 (id | composite-id),
 discriminator?,
 (version | timestamp)?,
 (property | many-to-one | one-to-one | component | dynamic-component |
  properties | any | map | set | list | bag | idbag |
  array | primitive-array)*,
 ((join*, subclass*) | joined-subclass* | union-subclass*),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 filter*
 resultset,
 (query | sql-query)
)
```

The <id> Element

All entities need to define their primary key in some way. Any class directly defined by the <class> element (not a derived or component class) must therefore have an <id> or a <composite-id> element to define this (see Table 7-4). Note that while it is not a requirement that your class implementation itself should implement the primary key attribute, it is certainly advisable. If you cannot alter your class design to accommodate this, you can instead use the getIdentifier() method on the Session object to determine the identifier of a persistent class independently.

Table 7-4. *The* <id> *Attributes*

Attribute	Values	Default	Description
access			Defines how the properties should be accessed: through field (directly), through property (calling the get/set methods), or through the name of a PropertyAccessor class to be used. The value from the <hibernate-mapping> element will be inherited if this is not specified.
column			The name of the column in the table containing the primary key. The value given in the name attribute will be used if this is not specified.
length			The column length to be used.

Attribute	Values	Default	Description
name			The name of the attribute in the class representing this primary key. If this is omitted, it is assumed that the class does not have an attribute directly representing this primary key. Naturally, the column attribute must be provided if the name attribute is omitted.
node			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features.
type			The Hibernate type of the column.
unsaved-value			The value that the attribute should take when an instance of the class has been created but not yet persisted to the database. This attribute is mandatory.

The <id> element requires a <generator> element to be specified, which defines how to generate a new primary key for a new instance of the class. The generator takes a class attribute, which defines the mechanism to be used. The class should be an implementation of org.hibernate.id.IdentifierGenerator. Optional <param> elements can be provided if the identifier needs additional configuration information, each having the following form:

```
<param name="parameter name">parameter value</param>
```

Hibernate provides several default IdentifierGenerator implementations, which can be referenced by convenient short names, as shown in Table 7-5. These are fairly comprehensive, so you are unlikely to need to implement your own IdentifierGenerator.

Table 7-5. *The Default* IdentiferGenerator *Implementations*

Short Name	Description
guid	Uses a database-generated "globally" unique identifier. This is not portable to databases that do not have a guid type. The specific implementation, and hence the quality of the uniqueness of this key, may vary from vendor to vendor.
hilo	Uses a database table and column to efficiently and portably maintain and generate identifiers that are unique to that database. The Hibernate int, short, and long types are supported.
identity	Supports the identity column type available in some, but not all, databases. This is therefore not a fully portable option. The Hibernate int, short, and long types are supported.
increment	Generates a suitable key by adding 1 to the current highest key value. Can apply to int, short, or long hibernate types. This only works if other processes are not permitted to update the table at the same time. If multiple processes are running, then depending on the constraints enforced by the database, the result may be an error in the application(s) or data corruption.
native	Selects one of sequence, identity, or hilo, as appropriate. This is a good compromise option since it uses the innate features of the database and is portable to most platforms. This is particularly appropriate if your code is likely to be deployed to a number of database implementations with differing capabilities.

Continued

Table 7-5. *Continued*

Short Name	Description
`seqhilo`	Uses a sequence to efficiently generate identifiers that are unique to that database. The Hibernate `int`, `short`, and `long` types are supported. This is not a portable technique (see `sequence`, following).
`sequence`	Supports the `sequence` column type (essentially a database-enforced increment) available in some, but not all, databases. This is, therefore, not a fully portable option. The Hibernate `int`, `short`, and `long` types are supported.
`uuid`	Attempts to portably generate a (cross-database) unique primary key. The key is composed of the local IP address, the startup time of the JVM (accurate to ¼ of a second), the system time, and a counter value (unique within the JVM). This cannot guarantee absolutely that a given key is unique, but it will be good enough for most clustering purposes.

The child elements of the `<id>` element are as follows:

```
(meta*, column*, type?, generator?)
```

While this is all rather complex, Listing 7-1 shows a typical `<id>` element from Chapter 3, which illustrates the simplicity of the usual case.

Listing 7-1. *A Typical* `<id>` *Element*

```
<id name="id" type="long" column="id">
   <generator class="native"/>
</id>
```

■ **Note** When the `<id>` element cannot be defined, a compound key can instead be defined using the `<composite-id>` element. This is provided purely to support existing database schemas. A new Hibernate project with a clean database design does not require this.

In addition to using the standard and custom generator types, you have the option of using the special `assigned` generator type. This allows you to explicitly set the identifier for the entities that you will be persisting—Hibernate will not then attempt to assign any identifier value to such an entity. If you use this technique, you will not be able to use the `saveOrUpdate()` method on a transient entity—instead, you will have to call the appropriate `save()` or `update()` method explicitly.

The <property> Element

While it is not absolutely essential, almost all classes will also maintain a set of properties in the database in addition to the primary key. These must be defined by a `<property>` element (see Table 7-6).

Table 7-6. *The* `<property>` *Attributes*

Attribute	Values	Default	Description
`access`			Defines how the properties should be accessed: through `field` (directly), through `property` (calling the get/set methods), or through the name of a `PropertyAccessor` class to be used. The value from the `<class>` element or `<hibernate-mapping>` element will be inherited if this is not specified.
`column`			The column in which the property will be maintained. If omitted, this will default to the name of the attribute; or it can be specified with nested `<column>` elements (see Listing 7-2).
`formula`			An arbitrary SQL query representing a computed property (i.e., one that is calculated dynamically, rather than represented in a column).
`index`			The name of an index to be maintained for the column.
`insert`	`true`, `false`	`true`	Specifies whether creation of an instance of the class should result in the column associated with this attribute being included in `insert` statements.
`lazy`	`true`, `false`	`false`	Defines whether lazy instantiation is used by default for this column.
`length`			The column length to be used.
`name`			The (mandatory) name of the attribute. This should start with a lowercase letter.
`node`			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features.
`not-null`	`true`, `false`	`false`	Specifies whether the column is permitted to contain `null` values.
`optimistic-lock`	`true`, `false`	`true`	Determines whether optimistic locking should be used when the attribute has been updated.
`precision`			Allows the precision (the number of digits) to be specified for numeric data.
`scale`			Allows the scale (the number of digits to the right of the decimal point) to be specified for numeric data.
`type`			The Hibernate type of the column.
`unique`	`true`, `false`	`false`	Indicates whether duplicate values are permitted for this column/attribute.
`unique-key`			Groups the columns together by this attribute value. Represents columns across which a unique key constraint should be generated (not yet supported in the schema generation).
`update`	`true`, `false`	`true`	Specifies whether changes to this attribute in instances of the class should result in the column associated with this attribute being included in `update` statements.

The child elements of the <property> element are as follows:

```
(meta*, (column | formula)*, type?)
```

Any element accepting a column attribute, as is the case for the <property> element, will also accept <column> elements in its place. For an example, see Listing 7-2.

Listing 7-2. *Using the* <column> *Element*

```
<property name="message"/>
   <column name="message" type="string"/>
</property>
```

This particular example does not really give us anything beyond the use of the column attribute directly; but the <column> element comes into its own with custom types and some of the more complex mappings that we will be looking into later in the chapter.

The <component> Element

The <component> element is used to map classes that will be represented as extra columns within a table describing some other class. We have already discussed how components fit in as a compromise between full entity types and mere value types.

The <component> element can take the attributes listed in Table 7-7.

Table 7-7. *The* <component> *Attributes*

Attribute	Values	Default	Description
access			Defines how the properties should be accessed: through field (directly), through property (calling the get/set methods), or through the name of a PropertyAccessor class to be used
class			The class that the parent class incorporates by composition
insert	true, false	true	Specifies whether creation of an instance of the class should result in the column associated with this attribute being included in insert statements
lazy	true, false	false	Defines whether lazy instantiation is used by default for this mapped entity
name			The name of the attribute (component) to be persisted
node			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features
optimistic-lock	true, false	true	Specifies the optimistic locking strategy to use
unique	true, false	false	Indicates that the values that represent the component must be unique within the table
update	true, false	true	Specifies whether changes to this attribute in instances of the class should result in the column associated with this attribute being included in update statements

The child elements of the <component> element are as follows:

```
(meta*,
 tuplizer*,
 parent?,
 (property | many-to-one | one-to-one |
  component | dynamic-component | any |
  map | set | list | bag |
  array | primitive-array)* )
```

We provide a full example of the use of the <component> element in the "Mapping Composition" section later in this chapter.

The <one-to-one> Element

The <one-to-one> element expresses the relationship between two classes, where each instance of the first class is related to a single instance of the second, and vice versa. Such a one-to-one relationship can be expressed either by giving each of the respective tables the same primary key values, or by using a foreign key constraint from one table onto a unique identifier column of the other. Table 7-8 shows the attributes that apply to the <one-to-one> element.

Table 7-8. *The* <one-to-one> *Attributes*

Attribute	Values	Default	Description
access			Specifies how the class member should be accessed: field for direct field access or attribute for access via the get and set methods.
cascade			Determines how changes to the parent entity will affect the linked relation.
check			The SQL to create a multirow check constraint for schema generation.
class			The property type of the attribute or field (if omitted, this will be determined by reflection).
constrained	true, false		Indicates that a foreign key constraint on the primary key of this class references the table of the associated class.
embed-xml	true, false		When using XML relational persistence, indicates whether the XML tree for the associated entity itself, or only its identifier, will appear in the generated XML tree.
entity-name			The entity name of the associated class.
fetch	join, select		The mode in which the element will be retrieved (outer join, a series of selects, or a series of subselects). Only one member of the enclosing class can be retrieved by outer join.
foreign-key			The name to assign to the foreign key enforcing the relationship.
formula			Allows the value to which the associated class maps its foreign key to be overridden using an SQL formula.

Continued

Table 7-8. *Continued*

Attribute	Values	Default	Description
lazy	true, false		Overrides the entity-loading mode.
name			Assigns a name to the entity (required in dynamic mappings).
node			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features.
outer join	true, false, auto		Specifies whether an outer join should be used.
property-ref			Specifies the column in the target entity's table that the foreign key references. If the referenced table's foreign key does not reference the primary key of the "many" end of the relationship, then property-ref can be used to specify the column that it references. This should only be the case for legacy designs—when creating a new schema, your foreign keys should always reference the primary key of the related table.

You would select a primary key association when you do not want an additional table column to relate the two entities. The master of the two entities takes a normal primary key generator, and its one-to-one mapping entry will typically have the attribute name and associated class specified only. The slave entity will be mapped similarly, but must have the constrained attribute setting applied to ensure that the relationship is recognized.

Because the slave class's primary key must be identical to that allocated to the master, it is given the special id generator type of foreign. On the slave end, the <id> and <one-to-one> elements will therefore look like this:

```
<id name="id" column="product">
   <generator class="foreign">
      <param name="property">campaign</param>
   </generator>
</id>

<one-to-one name="campaign"
   class="com.hibernatebook.xmlmapping.Campaign"
   constrained="true"/>
```

There are some limitations to this approach: it cannot be used on the receiving end of a many-to-one relationship (even when the "many" end of the association is limited by a unique constraint), and the slave entity cannot be the slave of more than one entity.

In these circumstances, you will need to declare the master end of the association as a uniquely constrained one-to-many association. The slave entity's table will then need to take a foreign key column associating it with the master's primary key. The property-ref attribute setting is used to declare this relationship, like so:

```
<one-to-one
   name="campaign"
   class="com.hibernatebook.xmlmapping.Campaign"
   property-ref="product"/>
```

The format used in this example is the most common. The body of the element consists of an infrequently used optional element:

```
(meta* | formula*)
```

We discuss the `<many-to-many>` element and the alternative approach of composition in some detail in the "Mapping Collections" section later in this chapter.

The <many-to-one> Element

The many-to-one association describes the relationship in which multiple instances of one class can reference a single instance of another class. This enforces a relational rule for which the "many" class has a foreign key into the (usually primary) unique key of the "one" class. Table 7-9 shows the attributes permissible for the `<many-to-one>` element.

Table 7-9. *The* `<many-to-one>` *Attributes*

Attribute	Values	Default	Description
`access`			Specifies how the class member should be accessed: `field` for direct field access, or `attribute` for access via the get and set methods.
`cascade`			Determines how changes to the parent entity will affect the linked relation.
`class`			The property type of the attribute or field (if omitted, this will be determined by reflection).
`column`			The column containing the identifier of the target entity (i.e., the foreign key from this entity into the mapped one).
`embed-xml`	`true`, `false`		When using XML relational persistence, indicates whether the XML tree for the associated entity itself, or only its identifier, will appear in the generated XML tree.
`entity-name`			The name of the associated entity.
`fetch`	`join`, `select`		The mode in which the element will be retrieved (`outer join`, a series of `selects`, or a series of `subselects`). Only one member of the enclosing class can be retrieved by `outer join`.
`foreign-key`			The name of the foreign key constraint to generate for this association.
`formula`			An arbitrary SQL expression to use in place of the normal primary key relationship between the entities.
`index`			The name of the index to be applied to the foreign key column in the parent table representing the "many" side of the association.

Continued

Table 7-9. *Continued*

Attribute	Values	Default	Description
insert	true, false	true	Indicates whether the field can be persisted. When set to false, this prevents inserts if the field has already been mapped as part of a composite identifier or some other attribute.
lazy	false, proxy, noproxy		Overrides the entity-loading mode.
name			The (mandatory) name of the attribute. This should start with a lowercase letter.
node			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features.
not-found	exception, ignore	exception	The behavior to exhibit if the related entity does not exist (either throw an exception or ignore the problem).
not-null	true, false	false	Specifies whether a not-null constraint should be applied to this column.
optimistic-lock	true, false	true	Specifies whether optimistic locking should be used.
outer-join	true, false, auto		Specifies whether an outer join should be used.
property-ref			Specifies the column in the target entity's table that the foreign key references. If the referenced table's foreign key does not reference the primary key of the "many" end of the relationship, then property-ref can be used to specify the column that it references. This should only be the case for legacy designs—when creating a new schema, your foreign keys should always reference the primary key of the related table.
unique	true, false	false	Specifies whether a unique constraint should be applied to the column.
unique-key			Groups the columns together by this attribute value. Represents columns across which a unique key constraint should be generated (not yet supported in the schema generation).
update	true, false	true	When set to false, prevents updates if the field has already been mapped elsewhere.

If a unique constraint is specified on a many-to-one relationship, it is effectively converted into a one-to-one relationship. This approach is preferred over creating a one-to-one association, both because it results in a simpler mapping and because it requires less intrusive changes to the database should it become desirable to relax the one-to-one association into a many-to-one.

This element has a small number of optional daughter elements—the <column> element will be required when a composite key has to be specified:

```
(meta*, (column | formula)*)
```

The following mapping illustrates the creation of a simple many-to-one association between a `User` class and an `Email` class: each user can have only one e-mail address—but an e-mail address can belong to more than one user.

```
<many-to-one
   name="email"
   class="com.hibernatebook.xmlmapping.Email"
   column="email"
   cascade="all" unique="true"/>
```

The simplest approach to creating a many-to-one relationship, as shown in the previous example, requires two tables and a foreign key dependency. An alternative is to use a link table to combine the two entities. The link table contains the appropriate foreign keys referencing the two tables associated with both of the entities in the association. The following code shows the mapping of a many-to-one relationship via a link table.

```
<join table="link_email_user" inverse="true" optional="false">
    <key column="user_id"/>
    <many-to-one name="email" column="email_id" not-null="true"/>
</join>
```

The disadvantage of the link table approach is its slightly poorer performance (it requires a join of three tables to retrieve the associations, rather than one). Its benefit is that it requires less extreme changes to the schema if the relationship is modified—typically, changes would be made to the link table, rather than to one of the entity tables.

The Collection Elements

These are the elements that are required for you to include an attribute in your class that represents any of the collection classes. For example, if you have an attribute of type `Set`, then you will need to use a `<bag>` or `<set>` element to represent its relationship with the database.

Because of the simplicity of the object-oriented relationship involved, where one object has an attribute capable of containing many objects, it is a common fallacy to assume that the relationship must be expressed as a one-to-many. In practice, however, this will almost always be easiest to express as a many-to-many relationship, where an additional link table closely corresponds with the role of the collection itself. See the "Mapping Collections" section later in this chapter for a more detailed illustration of this.

All the collection mapping elements share the attributes shown in Table 7-10.

Table 7-10. *The Attributes Common to the Collection Elements*

Attribute	Values	Default	Description
`access`			Specifies how the class member should be accessed: `field` for direct field access or `attribute` for access via the get and set methods.
`batch-size`			Specifies the number of items that can be batched together when retrieving instances of the class by identifier.

Continued

Table 7-10. *Continued*

Attribute	Values	Default	Description
cascade			Determines how changes to the parent entity will affect the linked relation.
catalog			The database catalog against which the queries should apply.
collection-type			The name of a UserCollectionType class describing the collection type to be used in place of the defaults.
check			The SQL to create a multirow check constraint for schema generation.
embed-xml	true, false		When using XML relational persistence, indicates whether the XML tree for the associated entity itself, or only its identifier, will appear in the generated XML tree.
fetch	join, select		The mode in which the element will be retrieved (outer-join, a series of selects, or a series of subselects). Only one member of the enclosing class can be retrieved by outer-join.
lazy	true, false		Can be used to disable or enable lazy fetching against the enclosing mapping's default.
mutable	true, false	true	Can be used to flag that a class is mutable (allowing Hibernate to make some performance optimizations when dealing with these classes).
name			The (mandatory) name of the attribute. This should start with a lowercase letter.
node			Specifies the name of the XML element or attribute that should be used by the XML relational persistence features.
optimistic-lock	true, false	true	Specifies the optimistic locking strategy to use.
outer-join	true, false, auto		Specifies whether an outer join should be used.
persister			Allows a custom ClassPersister object to be used when persisting this class.
schema			The database schema against which queries should apply.
subselect			A query to enforce a subselection of the contents of the underlying table. A class can only use a subselect if it is immutable and read-only (because the SQL defined here cannot be reversed). Generally, the use of a database view is preferable.
table			The name of the table in which the associated entity is stored.
where			An arbitrary SQL where clause limiting the linked entities.

The set Collection

A `set` collection allows collection attributes derived from the `Set` interface to be persisted.

In addition to the common collection mappings, the `<set>` element offers the `inverse`, `order-by`, and `sort` attributes, as shown in Table 7-11.

Table 7-11. *The Additional* `<set>` *Attributes*

Attribute	Values	Default	Description
`inverse`	`true`, `false`	`false`	Specifies that an entity is the opposite navigable end of a relationship expressed in another entity's mapping.
`order-by`			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the `set` collection.
`sort`			Specifies the collection class sorting to be used. The value can be `unsorted`, `natural`, or any `Comparator` class.

The child elements of the `<set>` element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 comment?,
 key,
 (element | one-to-many | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

The following code shows an implementation of mapping a set of strings into a property called `titles`:

```
<set name="titles" table="nameset">
   <key column="titleid"/>
   <element type="string" column="name" not-null="true"/>
</set>
```

A typical implementation, however, maps other entities into the collection. Here we map `Phone` entities from the "many" side of a one-to-many association into a `Set` property, called `phoneNumbers`, that belongs to a `User` entity:

```
<set name="phoneNumbers">
   <key column="aduser"/>
   <one-to-many class="sample.Phone"/>
</set>
```

If the Phone class contains a reference to a User object, it is not automatically clear whether this constitutes a pair of unrelated associations or two halves of the same association—a bidirectional association. When a bidirectional association is to be established, one side must be selected as the owner (in a one-to-many or many-to-one association, it must always be the "many" side), and the other will be marked as being the inverse half of the relationship. See the discussion of unidirectional and bidirectional associations at the end of Chapter 4. The following code shows a mapping of a one-to-many relationship as a reverse association.

```
<set name="phoneNumbers" inverse="true">
   <key column="aduser"/>
   <one-to-many class="sample.Phone"/>
</set>
```

The list Collection

A list collection allows collection attributes derived from the List interface to be persisted.

In addition to the common collection mappings, the <list> element offers the inverse attribute, as shown in Table 7-12.

Table 7-12. *The Additional* <list> *Attribute*

Attribute	Values	Default	Description
inverse	true, false	false	Specifies that an entity is the opposite navigable end of a relationship expressed in another entity's mapping

The child elements of the <list> element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 comment?,
 key,
 (index | list-index),
 (element | one-to-many | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of a `list` mapping is as follows:

```
<list name="list" table="namelist">
   <key column="fooid"/>
   <index column="position"/>
   <element type="string" column="name" not-null="true"/>
</list>
```

The idbag Collection

An `idbag` collection allows for appropriate use of collection attributes derived from the `List` interface. A `bag` data structure permits unordered storage of unordered items, and permits duplicates. Because the collection classes do not provide a native `bag` implementation, classes derived from the `List` interface tend to be used as a substitute. The imposition of ordering imposed by a `list` is not itself a problem, but the implementation code can become dependent upon the ordering information.

`idbag` usually maps to a `List`. However, by managing its database representation with a surrogate key, you can make the performance of updates and deletions of items in a collection defined with `idbag` dramatically better than with an unkeyed `bag` (described at the end of this section). Hibernate does not provide a mechanism for obtaining the identifier of a row in the `bag`.

In addition to the common collection mappings, the `<idbag>` element offers the `order-by` element, as shown in Table 7-13.

Table 7-13. *The Additional* `<idbag>` *Attribute*

Attribute	Values	Default	Description
`order-by`			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the collection

The child elements of the `<idbag>` element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 comment?,
 collection-id,
 key,
 (element | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of an idbag mapping is as follows:

```
<idbag name="idbag" table="nameidbag">
   <collection-id column="id" type="int">
      <generator class="native"/>
   </collection-id>

   <key column="fooid"/>
   <element type="string" column="name" not-null="true"/>
</idbag>
```

The map Collection

A map collection allows collection attributes derived from the Map interface to be persisted.

In addition to the common collection mappings, the <map> element offers the inverse, order-by, and sort attributes, as shown in Table 7-14.

Table 7-14. *The Additional* <map> *Attributes*

Attribute	Values	Default	Description
inverse	true, false	false	Specifies that this entity is the opposite navigable end of a relationship expressed in another entity's mapping
order-by			Specifies an arbitrary SQL order by clause to constrain the results returned by the SQL query that populates the map
sort		unsorted	Specifies the collection class sorting to be used. The value can be unsorted, natural, or any Comparator class

The child elements of the <map> element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 comment?,
 key,
 (map-key | composite-map-key | map-key-many-to-many |
  index | composite-index | index-many-to-many |
  index-many-to-any),
 (element | one-to-many | many-to-many | composite-element |
  many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of the mapping is as follows:

```
<map name="map" table="namemap">
   <key column="fooid"/>
   <index column="name" type="string"/>
   <element column="value" type="string" not-null="true"/>
</map>
```

The bag Collection

If your class represents data using a class derived from the `List` interface, but you do not want to maintain an index column to keep track of the order of items, you can optionally use the `bag` collection mapping to achieve this. The order in which the items are stored and retrieved from a `bag` is completely ignored.

Although the `bag`'s table does not contain enough information to determine the order of its contents prior to persistence into the table, it *is* possible to apply an `order by` clause to the SQL used to obtain the contents of the `bag` so that it has a natural sorted order as it is acquired. This will not be honored at other times during the lifetime of the object.

If the `<bag>` elements lack a proper key, there will be a performance impact that will manifest itself when update or delete operations are performed on the contents of the `bag`.

In addition to the common collection mappings, the `<bag>` element therefore offers the `order-by` as well as the `inverse` attribute, as shown in Table 7-15.

Table 7-15. *The Additional* `<bag>` *Attributes*

Attribute	Values	Default	Description
`inverse`	`true`, `false`	`false`	Specifies that an entity is the opposite navigable end of a relationship expressed in another entity's mapping
`order-by`			Specifies an arbitrary SQL `order by` clause to constrain the results returned by the SQL query that populates the collection

The child elements of the `<bag>` element are as follows:

```
(meta*,
 subselect?,
 cache?,
 synchronize*,
 comment?,
 key,
 (element | one-to-many | many-to-many |
  composite-element | many-to-any),
 loader?,
 sql-insert?,
 sql-update?,
 sql-delete?,
 sql-delete-all?,
 filter*)
```

A typical implementation of a bag mapping is as follows:

```
<bag name="bag" table="namebag">
   <key column="fooid"/>
   <element column="value" type="string" not-null="true"/>
</bag>
```

Mapping Simple Classes

Figure 7-1 shows the class diagram and entity relationship diagram for a simple class. They are as straightforward as you would expect.

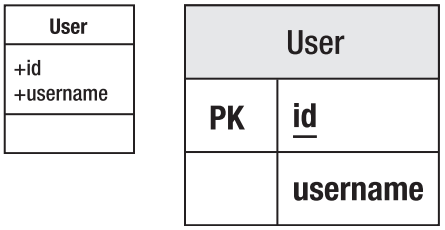

Figure 7-1. *Representing a simple class*

The elements discussed so far are sufficient to map a basic class into a single table, as shown in Listing 7-3.

Listing 7-3. *A Simple Class to Represent a User*

```
package com.hibernatebook.xmlmapping;

public class User {

   public User(String username) {
      this.username = username;
   }

   User() {
   }

   public int getId() {
      return id;
   }

   public String getUsername() {
      return username;
   }
```

```
   public void setId(int id) {
      this.id = id;
   }

   public void setUsername(String username) {
      this.username = username;
   }

   // We will map the id to the table's primary key
   private int id = -1;

   // We will map the username into a column in the table
   private String username;
}
```

It's pretty easy to see that we might want to represent the class in Listing 7-3 in a table with the format shown in Table 7-16.

Table 7-16. *Mapping a Simple Class to a Simple Table*

Column	Type
Id	Integer
Username	Varchar(32)

The mapping between the two is, thus, similarly straightforward:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="book.hibernatebook.chapter06.User">

      <id name="id" type="int">
         <generator class="native"/>
      </id>

      <property name="username" type="string" length="32"/>

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

Aside from the very limited number of properties maintained by the class, this is a pretty common mapping type, so it is reassuring to see that it can be managed with a minimal number of elements (<hibernate-mapping>, <class>, <id>, <generator>, and <property>).

Mapping Composition

Figure 7-2 shows the class diagram and the entity relationship diagram for a composition relationship between two classes. Here, the `Advert` class is composed of a `Picture` class in addition to its normal value types.

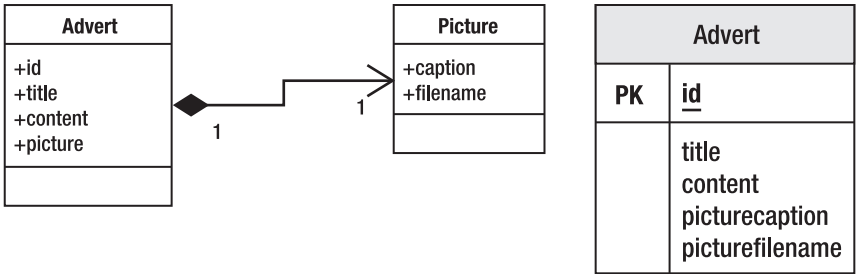

Figure 7-2. *Representing composition*

Composition is the strongest form of aggregation—in which the life cycle of each object is dependent upon the life cycle of the whole. Although Java does not make the distinction between other types of aggregation and composition, it becomes relevant when we choose to store the components in the database, because the most efficient and natural way to do this is to store them in the same table.

In our example, we will look at an `Advert` class that has this relationship with a `Picture` class. The idea is that our advert is always going to be associated with an illustration (see Listings 7-4 and 7-5). In these circumstances, there is a clear one-to-one relationship that could be represented between two distinct tables, but which is more efficiently represented with one.

Listing 7-4. *The Class Representing the Illustration*

```
package com.hibernatebook.xmlmapping;

public class Picture {
   public Picture(String caption, String filename) {
      this.caption = caption;
      this.filename = filename;
   }

   Picture() {
   }

   public String getCaption() {
      return this.caption;
   }
```

```
   public String getFilename() {
      return this.filename;
   }

   public void setCaption(String title) {
      this.caption = title;
   }

   public void setFilename(String filename) {
      this.filename = filename;
   }

   private String caption;
   private String filename;
}
```

Listing 7-5. *The Class Representing the Advert*

```
package com.hibernatebook.xmlmapping;

public class Advert {
   public Advert(String title, String content, Picture picture) {
      this.title = title;
      this.content = content;
      this.picture = picture;
   }

   Advert() {
   }

   public int getId() {
      return id;
   }

   public String getTitle() {
      return this.title;
   }

   public String getContent() {
      return this.content;
   }

   public Picture getPicture() {
      return this.picture;
   }
```

```
    public void setId(int id) {
        this.id = id;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public void setPicture(Picture picture) {
        this.picture = picture;
    }

    private int id = -1;
    private String title;
    private String content;
    private Picture picture;
}
```

Again, Hibernate manages to express this simple relationship with a correspondingly simple mapping file. We introduce the `component` entity for this association. Here it is in use:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<class name="com.hibernatebook.xmlmapping.Advert">
    <id name="id" type="int">
        <generator class="native"/>
    </id>
    <property name="title" type="string" length="255"/>
    <property name="content" type="text"/>
    <component name="picture" class="com.hibernatebook.xmlmapping.Picture">
        <property name="caption" type="string" length="255"/>
        <property name="filename" type="string" length="32"/>
    </component>
</class>
```

In this example, we use the `<property>` element to describe the relationship between `Picture` and its attributes. In fact, this is true of all of the rest of the elements of `<class>`—a `<component>` element can even contain more `<component>` elements. Of course, this makes perfect sense, since a component usually corresponds with a Java class.

Mapping Other Associations

In Figure 7-3, the `Advert` class includes an instance of a `Picture` class. The relationship in the tables is represented with the `Picture` table having a foreign key onto the `Advert` table.

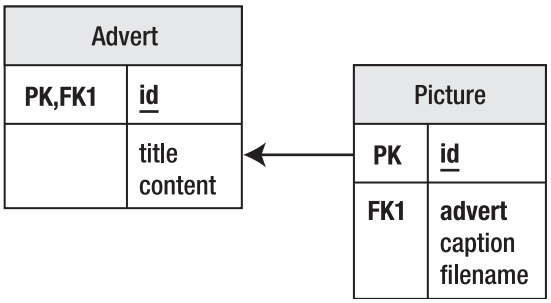

Figure 7-3. *Mapping an aggregation or composition relationship*

A one-to-one correspondence does not absolutely require you to incorporate both parties into the same table. There are often good reasons not to. For instance, in the `Picture` example, it is entirely possible that while the initial implementation will permit only one `Picture` per `Advert`, a future implementation will relax this relationship. Consider this scenario from the perspective of the database for a moment (see Table 7-17).

Table 7-17. *The* `Advert` *Table*

ID	Title	Contents	PictureCaption	PictureFilename
1	Bike	Bicycle for sale	My bike (you can ride it if you like)	`advert001.jpg`
2	Sofa	Sofa, comfy but used	Chesterfield sofa	`advert002.jpg`
3	Car	Shabby MGF for sale	MGF VVC (BRG)	`advert003.jpg`

If we want to allow the advert for the sofa to include another picture, we would have to duplicate some of the data, or include null columns. It would probably be preferable to set up a pair of tables: one to represent the adverts, and one to represent the distinct tables (as shown in Tables 7-18 and 7-19).

Table 7-18. *The Refined* `Advert` *Table*

ID	Title	Contents
1	Bike	Bicycle for sale
2	Sofa	Sofa, comfy but used
3	Car	Shabby MGF for sale

Table 7-19. *The* `Picture` *Table*

ID	Advert	Caption	Filename
1	1	My bike (you can ride it if you like)	`advert001.jpg`
2	2	Chesterfield sofa	`advert002.jpg`
3	3	MGF VVC (BRG)	`advert003.jpg`

If we decide (considering the database only) to allow additional pictures, we can then include extra rows in the `Picture` table without duplicating any data unnecessarily (see Table 7-20).

Table 7-20. *The* `Picture` *Table with Multiple Pictures per Advert*

ID	Advert	Caption	Filename
1	1	My bike (you can ride it if you like)	`advert001.jpg`
2	2	Chesterfield sofa	`advert002.jpg`
3	2	Back of sofa	`advert003.jpg`
4	3	MGF VVC (BRG)	`advert004.jpg`

With the single `Advert` table, the query to extract the data necessary to materialize an instance of the `Advert` consists of something like this:

```
select id,title,contents,picturecaption,picturefilename from advert where id = 1
```

It is obvious here that a single row will be returned, since we are carrying out the selection on the primary key.

Once we split things into two tables, we have a slightly more ambiguous pair of queries:

```
select id,title,contents from advert where id = 1
select id,caption,filename from picture where advert = 1
```

While Hibernate is not under any particular obligation to use this pair of SQL instructions to retrieve the data (it could reduce it to a join on the table pair), it is the easiest way of thinking about the data we are going to retrieve. While the first query of the two is required to return a single row, this is not true for the second query—if we have added multiple pictures, we will get multiple rows back.

In these circumstances, there is very little difference between a one-to-one relationship and a one-to-many relationship, except from a business perspective. That is to say, we choose not to associate an advert with multiple pictures, even though we have that option.

This, perhaps, explains why the expression of a one-to-one relationship in Hibernate is usually carried out via a many-to-one mapping. If you do not find that persuasive, remember that a foreign key relationship, which is the relationship that the `advert` column in the `Picture` table has with the `id` column in the `Advert` table, is a many-to-one relationship between the entities.

In our example, the `Picture` table will be maintaining the `advert` column as a foreign key into the `Advert` table, so this must be expressed as a many-to-one relationship with the `Advert` object (see Listing 7-6).

Listing 7-6. *The New Picture Mapping*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<class name="com.hibernatebook.xmlmapping.Picture">
    <id name="id" type="int">
        <generator class="native"/>
    </id>
    <many-to-one
      name="advert"
      class="com.hibernatebook.xmlmapping.Advert"
      column="advert"/>
    <property name="caption" type="string" length="255"/>
    <property name="filename" type="string" length="32"/>
</class>
```

If you still object to the many-to-one relationship, you will probably find it cathartic to note that we have explicitly constrained this relationship with the `unique` attribute. You will also find it reassuring that in order to make navigation possible directly from the `Advert` to its associated `Picture`, we can in fact use a one-to-one mapping entry. We need to be able to navigate in this direction because we expect to retrieve adverts from the database, and then display their associated pictures (see Listing 7-7).

Listing 7-7. *The Revised* `Advert` *Mapping*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<class name="com.hibernatebook.xmlmapping.Advert">
    <id name="id" type="int">
        <generator class="native"/>
    </id>
    <property name="title" type="string" length="255"/>
    <property name="content" type="text"/>
    <one-to-one name="picture"
                class="com.hibernatebook.xmlmapping.Picture"
                property-ref="picture">
</class>
```

Now that we have seen how one-to-one and many-to-one relationships are expressed, we will see how a many-to-many relationship can be expressed.

Mapping Collections

In Figure 7-4, we show the `User` objects as having an unknown number of `Advert` instances. In the database, this is then represented with three tables, one of which is a link table between the two entity tables.

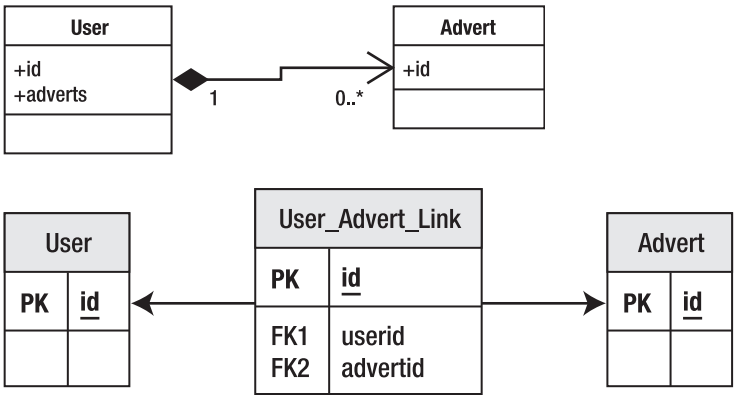

Figure 7-4. *Mapping collections*

The Java collection classes provide the most elegant mechanism for expressing the "many" end of a many-to-many relationship in our own classes:

```
public Set getAdverts();
```

If we use generics, we can give an even more precise specification:

```
public Set<Advert> getAdverts();
```

■ **Note** A lot of legacy code will not use generics. However, if you have the opportunity you should do so, as it allows you to make this sort of distinction clear at the API level, instead of at the documentation level. Hibernate 3 is compatible with Java 5 generics.

Of course, we can place values (of `Object` type) into collections as well as entities, and Java 5 introduced autoboxing so that we have the illusion of being able to place primitives into them as well.

```
List<Integer> ages = getAges();
int first = ages.get(0);
```

The only catch with collection mapping is that an additional table may be required to correctly express the relationship between the owning table and the collection. Table 7-21 shows how it should be done; the entity table contains only its own attributes.

Table 7-21. *The* `Entity` *Table*

ID	Name
1	Entity 1

A separate collection table, on the other hand, contains the actual values (see Table 7-22). In this case, we are linking a `List` to the owning entity, so we need to include a column to represent the position of the values in the list, as well as the foreign key into the owning entity and the column for the actual values that are contained within the collection.

Table 7-22. `ListTable`

entityid	positionInList	listValue
1	1	Good
1	2	Bad
1	3	Indifferent

In a legacy schema, you may quite often encounter a situation in which all the values have been retained within a single table (see Table 7-23).

Table 7-23. `EntityTable`

ID	Name	positionInList	listValue
1	Entity 1	1	Good
1	Entity 1	2	Bad
1	Entity 1	3	Indifferent

It should be obvious that this is not just poor design from Hibernate's perspective—it's also bad relational design. The values in the entity's `name` attribute have been duplicated needlessly, so this is not a properly normalized table. We also break the foreign key of the table, and need to form a compound key of `id` and `positionInList`. Overall, this is a poor design, and we encourage you to use a second table if at all possible. If you must work with such an existing design, see Chapter 13 for some techniques for approaching this type of problem.

If your collection is going to contain entity types instead of value types, the approach is essentially the same, but your second table will contain keys into the second entity table instead of value types. This changes the combination of tables into the situation shown in the entity relationship diagram (see Figure 7-4), in which we have a link table joining two major tables into a many-to-many relationship. This is a very familiar pattern in properly normalized relational schemas.

The following code shows a mapping of a `Set` attribute representing the adverts with which the `User` class is associated:

```
<set name="adverts"
     table="user_advert_link"
     cascade="save-update">
   <key column="userid"/>
   <many-to-many
      class="com.hibernatebook.xmlmapping.Advert"
      column="advertid"/>
</set>
```

Hibernate's use of collections tends to expose the lazy loading issues more than most other mappings. If you enable lazy loading, the collection that you retrieve from the session will be a proxy implementing the relevant collection interface (in our example, `Set`), rather than one of the usual Java concrete collection implementations.

This allows Hibernate to retrieve the contents of the collection only as they are required by the user. If you load an entity, consult a single item from the collection, and then discard it, often only a handful of SQL operations will be required. If the collection in question represents hundreds of entity instances, the performance advantages of lazy loading (compared with the massive task of reading in *all* of the entities concerned) are massive.

However, you will need to ensure that you do not try to access the contents of a lazily loaded collection at a time when it is no longer associated with the session, unless you can be certain that the contents of the collection that you are accessing have already been loaded.

Mapping Inheritance Relationships

Figure 7-5 shows a simple class hierarchy. The superclass is `Advert`, and there are two classes derived from this: a `Personal` class to represent personal advertisements and a `Property` class to represent property advertisements.

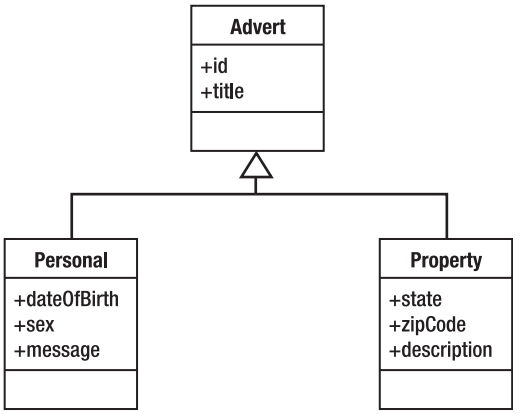

Figure 7-5. *A simple inheritance hierarchy*

Hibernate can represent inheritance relationships in a relational schema in three ways, each mapped in a slightly different way. These are as follows:

- One table for each concrete class implementation
- One table for each subclass (including interfaces and abstract classes)
- One table for each class hierarchy

Each of these techniques has different costs and benefits, so we will show you an example mapping from each and discuss some of these issues.

One Table per Concrete Class

This approach is the easiest to implement. You map each of the concrete classes as normal, writing mapping elements for each of its persistent properties (including those that are inherited). No mapping files are required for interfaces and abstract classes.

Figure 7-6 shows the schema required to represent the hierarchy from Figure 7-5 using this technique.

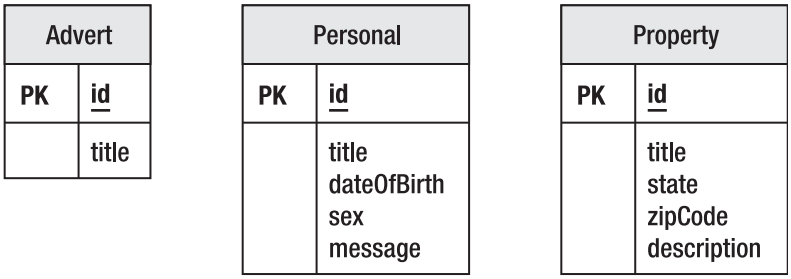

Figure 7-6. *Mapping one table per concrete class*

While this is easy to create, there are several disadvantages; the data belonging to a parent class is scattered across a number of different tables, so a query couched in terms of the parent class is likely to cause a large number of select operations. It also means that changes to a parent class can touch an awful lot of tables. We suggest that you file this approach under "quick-and-dirty solutions."

Listing 7-8 demonstrates how a derived class (`Property`) can be mapped to a single table independently of its superclass (`Advert`).

Listing 7-8. *Mapping a* `Property` *Advert with the One-Table-per-Concrete-Class Approach*

```
<hibernate-mapping>
   <class name="com.hibernatebook.xmlmapping.Property">
      <id name="id" type="int">
          <generator class="native"/>
      </id>
      <property name="title" type="string" length="255"/>
```

```
      <property name="state" type="string"/>
      <property name="zipCode" type="string"/>
      <property name="description" type="string"/>
   </class>
</hibernate-mapping>
```

One Table per Subclass

A slightly more complex mapping is to provide one table for each class in the hierarchy, including the abstract and interface classes. The pure "is a" relationship of our class hierarchy is then converted into a "has a" relationship for each entity in the schema.

Figure 7-7 shows the schema required to represent the hierarchy from Figure 7-5 using this technique.

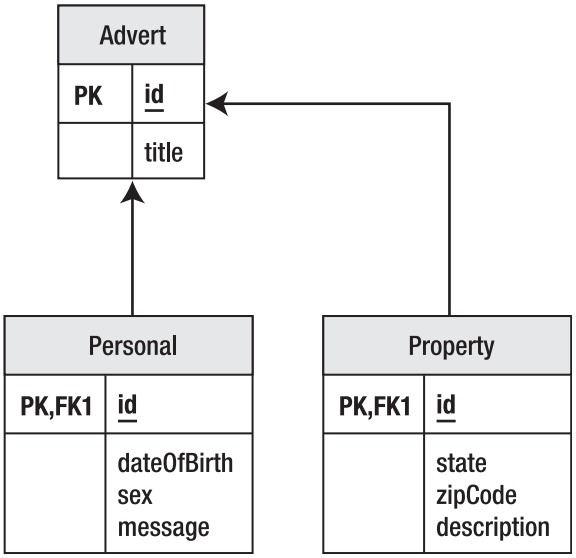

Figure 7-7. *Mapping one table per subclass*

We like this approach, as it is conceptually easy to manage, does not require complex changes to the schema when a single parent class is modified, and is similar to how most JVMs manage the same data behind the scenes.

The disadvantage of this approach is that while it works well from an object-oriented point of view, and is correct from a relational point of view, it can result in poor performance. As the hierarchy grows, the number of joins required to construct a leaf class also grows.

The technique works well for shallow inheritance hierarchies. Deep inheritance hierarchies are often a symptom of poorly designed code, so you may want to reconsider your application architecture before abandoning this technique. In our opinion, it should be preferred until performance issues are substantially proven to be an issue.

Listing 7-9 shows how you can map a derived class (`Property`) as a table joined to another representing the superclass (`Advert`).

Listing 7-9. *Mapping a* `Property` *Advert with the One-Table-per-Subclass Approach*

```
<hibernate-mapping>
   <joined-subclass
      name="com.hibernatebook.xmlmapping.Property"
      extends="com.hibernatebook.xmlmapping.Advert">

      <key column="advertid"/>

      <property name="state" type="string"/>
      <property name="zipCode" type="string"/>
      <property name="description" type="string"/>
   </joined-subclass>
</hibernate-mapping>
```

Note in the mapping that we replace `class` with `joined-subclass` to associate our mapping explicitly with the parent. You specify the entity that is being extended and replace the `id` and `title` classes from the subclass with a single key element that maps the foreign key column to the parent class table's primary key. Otherwise, the `<joined-subclass>` element is virtually identical to the `<class>` element. Note, however, that a `<joined-subclass>` cannot contain `<subclass>` elements and vice versa—the two strategies are not compatible.

One Table per Class Hierarchy

The last of the inheritance mapping strategies is to place each inheritance hierarchy in its own table. The fields from each of the child classes are added to this table, and a discriminator column contains a key to identify the base type represented by each row in the table.

Figure 7-8 shows the schema required to represent the hierarchy from Figure 7-5 using this technique.

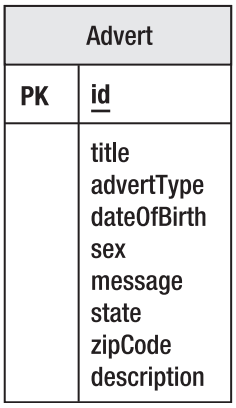

Figure 7-8. *Mapping one table per hierarchy*

This technique offers the best performance—for simple queries on simple classes even in the deepest of inheritance hierarchies, a single select may suffice to gather all the fields to populate the entity.

Conversely, this is not a satisfying representation of the attribute. Changes to members of the hierarchy will usually require a column to be altered, added, or deleted from the table. This will often be a very slow operation. As the hierarchy grows (horizontally as well as vertically), so too will the number of columns required by this table.

Each mapped subclass must specify the class that it extends and a value that can be used to discriminate this subclass from the other classes held in the same table. Thus, this is known as the discriminator value, and is mapped with a `discriminator-value` attribute in the `<subclass>` element (see Listing 7-10).

Listing 7-10. *Mapping a* `Property` *Advert with the One-Table-per-Class-Hierarchy Approach*

```
<hibernate-mapping>
   <subclass
      name="com.hibernatebook.xmlmapping.Property"
      extends="com.hibernatebook.xmlmapping.Advert"
      discriminator-value="property">

      <property name="state" type="string"/>
      <property name="zipCode" type="string"/>
      <property name="description" type="string"/>
   </subclass>
</hibernate-mapping>
```

Note that this also requires the specification of a discriminator column for the root of the class hierarchy, from which the discriminator values identifying the types of the child classes can be obtained (see Listing 7-11).

Listing 7-11. *The Addition to* `Advert.hbm.xml` *Required to Support a One-Table-per-Class-Hierarchy Approach*

```
<discriminator column="advertType" type="string"/>
```

A subclass mapping cannot contain `<joined-subclass>` elements and vice versa—the two strategies are not compatible.

More Exotic Mappings

The Hibernate mapping DTD is large. We have discussed the core set of mappings that you will use on a day-to-day basis; but before we move on, we will take a very quick tour around four of the more interesting remaining mapping types.

The any Tag

The `any` tag represents a polymorphic association between the attribute and several entity classes. The mapping is expressed in the schema with a column to specify the type of the related entity, and then columns for the identifier of the related entity.

Because a proper foreign key cannot be specified (being dependent upon multiple tables), this is not the generally recommended technique for making polymorphic associations. When possible, use the techniques described in the previous "Mapping Inheritance Relationships" section.

The array Tag

The `array` tag represents the innate array feature of the Java language. The syntax of this is virtually identical to that used for the `List` collection class, and we recommend the use of `List` except when primitive values are to be stored, or when you are constrained by an existing application architecture to work with arrays.

The <dynamic-component> Element

While the full-blown `dynamic` class approach (discussed briefly in the "Entities" section at the beginning of the chapter) is really only suitable for prototyping exercises, the dynamic component technique allows some of that flexibility in a package that reflects some legitimate techniques.

The `<dynamic-component>` element permits you to place any of the items that can be mapped with the normal `<component>` element into a map with a given key. For example, we could obtain and combine several items of information relating to an entity's ownership into a single `Map` with named elements, as follows:

```
<dynamic-component name="ownership">
   <property name="user" type="string" column="user"/>
   <many-to-one
      name="person"
      class="com.hibernatebook.xmlmapping.Person"
      column="person_id"/>
</dynamic-component>
```

The code to access this information in the entity is then very familiar:

```
Map map = entity.getOwnership();
System.out.println(map.get("user"));
System.out.println(map.get("person"));
```

The output would then be as follows:

```
dcminter
person: { "Dave Minter", 33, "5'10" }
```

Summary

This chapter has covered the data types supported by Hibernate 3: entities, values, and components. You have seen how all three can be expressed in a mapping file, and how each relates to the underlying database schema. We have listed the attributes available to the major mapping elements, and we have discussed some detailed examples of the elements that you will use most frequently when working with Hibernate.

In the next chapter, we will look at how a client application communicates with the database representation of the entities by using the `Session` object.