The persistence lifecycle

Since Hibernate is a transparent persistence mechanism—classes are unaware of their own persistence capability—it's possible to write application logic that is unaware of whether the objects it operates on represent persistent state or temporary state that exists only in memory. The application shouldn't necessarily need to care that an object is persistent when invoking its methods.

However, in any application with persistent state, the application must interact with the persistence layer whenever it needs to propagate state held in memory to the database (or vice versa). To do this, you call Hibernate's persistence manager and query interfaces. When interacting with the persistence mechanism that way, it's necessary for the application to concern itself with the state and lifecycle of an object with respect to persistence. We'll refer to this as the *persistence lifecycle*. Different ORM implementations use different terminology and define different states and state transitions for the persistence lifecycle. Moreover, the object states used internally might be different from those exposed to the client application.

Hibernate defines only three states, hiding the complexity of its internal implementation from the client code. In this section, we explain these three states: *transient*, *persistent*, and *detached*.

Let's look at these states and their transitions in a state chart, shown in figure 4.1. You can also see the method calls to the persistence manager that trigger transitions. We discuss this chart in this section; refer to it later whenever you need an overview.

In its lifecycle, an object can transition from a transient object to a persistent object to a detached object. Let's take a closer look at each of these states.

4.1.1 Transient objects

In Hibernate, objects instantiated using the `new` operator aren't immediately persistent.

Their state is *transient*, which means they aren't associated with any database table row, and so their state is lost as soon as they're dereferenced (no longer referenced by any other object) by the application. These objects have a lifespan that

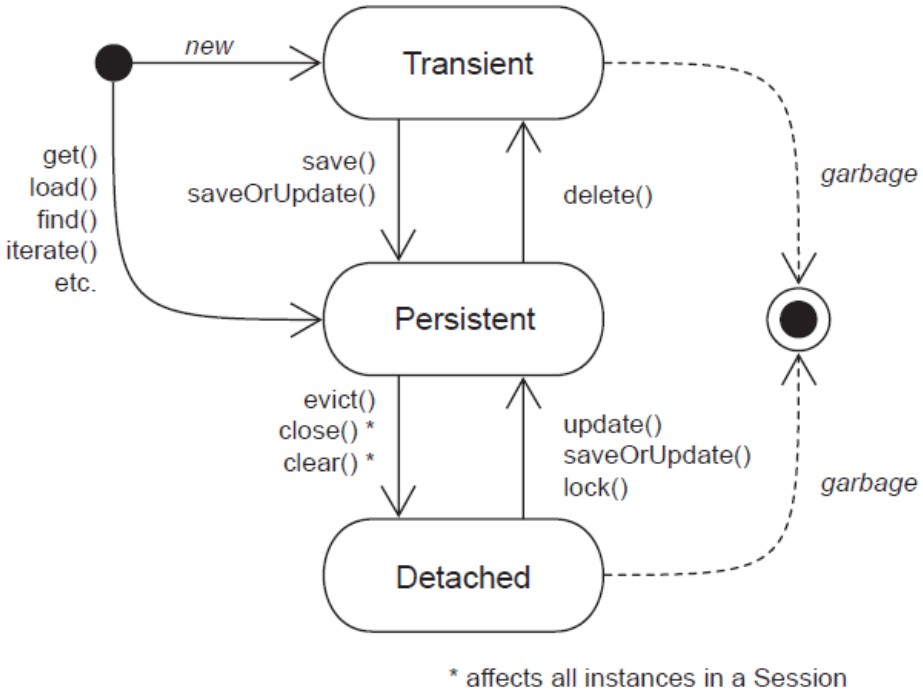

Figure 4.1 States of an object and transitions in a Hibernate application

effectively ends at that time, and they become inaccessible and available for garbage collection. Hibernate considers all transient instances to be nontransactional; a modification to the state of a transient instance isn't made in the context of any transaction. This means Hibernate doesn't provide any rollback functionality for transient objects. (In fact, Hibernate doesn't roll back any object changes, as you'll see later.) Objects that are referenced only by other transient instances are, by default, also transient. For an instance to transition from transient to persistent state

requires either a `save()` call to the persistence manager or the creation of a reference from an already persistent instance.

4.1.2 Persistent objects

A persistent instance is any instance with a *database identity*. Persistent instances might be objects instantiated by the application and then made persistent by calling the `save()` method of the persistence manager (the Hibernate `Session`, discussed in more detail later in this chapter). Persistent instances are then associated with the persistence manager. They might even be objects that became persistent when a reference was created from another persistent object already associated with a persistence manager. Alternatively, a persistent instance might be an instance retrieved from the database by execution of a query, by an identifier lookup, or by navigating the object graph starting from another persistent instance. In other words, persistent instances are always associated with a `Session` and are *transactional*.

Persistent instances participate in transactions—their state is synchronized with the database at the end of the transaction. When a transaction commits, state held in memory is propagated to the database by the execution of SQL `INSERT`, `UPDATE`, and `DELETE` statements. This procedure might also occur at other times. For example, Hibernate might synchronize with the database before execution of a query. This ensures that queries will be aware of changes made earlier during the transaction.

We call a persistent instance *new* if it has been allocated a primary key value but has not yet been inserted into the database. The new persistent instance will remain "new" until synchronization occurs. Of course, you don't update the database row of every persistent object in memory at the end of the transaction. ORM software must have a strategy for detecting which persistent objects have been modified by the application in the transaction.

We call this *automatic dirty checking* (an object with modifications that haven't yet been propagated to the database is considered *dirty)*. Again, this state isn't visible to the application. We call this feature *transparent transaction-level write-behind*, meaning that Hibernate propagates state changes to the database as late as possible but hides this detail from the application.

Hibernate can detect exactly which attributes have been modified, so it's possible to include only the columns that need updating in the SQL `UPDATE` statement.

This might bring performance gains, particularly with certain databases. However, it isn't usually a significant difference, and, in theory, it could harm performance in some environments. So, by default, Hibernate includes all columns in the SQL `UPDATE` statement (hence, Hibernate can generate this basic SQL at startup, not at runtime). If you only want to update modified columns, you can enable dynamic SQL generation by setting `dynamic-update="true"` in a class mapping. (Note that this feature is extremely difficult to implement in a handcoded persistence layer.) Finally, a persistent instance may be made transient via a `delete()` call to the persistence manager API, resulting in deletion of the corresponding row of the database table.

4.1.3 Detached objects

When a transaction completes, the persistent instances associated with the persistence manager still exist. (If the transaction were successful, their in-memory state will have been synchronized with the database.) In ORM implementations with *process-scoped identity* (see the following sections), the instances retain their association to the persistence manager and are still considered persistent.

In the case of Hibernate, however, these instances lose their association with the persistence manager when you `close()` the `Session`. We refer to these objects as *detached*, indicating that their state is no longer guaranteed to be synchronized with database state; they're no longer under the management of Hibernate. However, they still contain persistent data (that may possibly soon be stale). It's possible (and common) for the application to retain a reference to a detached object outside of a transaction (and persistence manager). Hibernate lets you reuse these instances in a new transaction by reassociating them with a new persistence manager. (After reassociation, they're considered persistent.) This feature has a deep impact on how multitiered applications may be designed. The ability to return objects from one transaction to the presentation layer and later reuse them in a new transaction is one of Hibernate's main selling points. Hibernate also provides an explicit detachment operation: the `evict()` method of the `Session`. However, this method is typically used only for cache management (a performance consideration). It's *not* normal to perform detachment explicitly. Rather, all objects retrieved in a transaction become detached when the `Session` is closed or when they're serialized (if they're passed remotely, for example). So, Hibernate doesn't need to provide functionality for controlling detachment of *subgraphs*.

Instead, the application can control the depth of the fetched subgraph (the instances that are currently loaded in memory) using the query language or explicit graph navigation. Then, when the `Session` is closed, this entire subgraph (all objects associated with a persistence manager) becomes detached.
Let's look at the different states again but this time consider the *scope of object identity*.

4.1.4 The scope of object identity

As application developers, we identify an object using Java object identity `(a==b)`.
So, if an object changes state, is its Java identity guaranteed to be the same in the new state? In a layered application, that might not be the case.
In order to explore this topic, it's important to understand the relationship between Java identity, `a==b`, and database identity, `a.getId().equals( b.getId() )`.
Sometimes both are equivalent; sometimes they aren't. We refer to the conditions under which Java identity is equivalent to database identity as the *scope of object identity*.
For this scope, there are three common choices:

- A primitive persistence layer with *no identity scope* makes no guarantees that if a row is accessed twice, the same Java object instance will be returned to the application. This becomes problematic if the application modifies two different instances that both represent the same row in a single transaction (how do you decide which state should be propagated to the database?).
- A persistence layer using *transaction-scoped identity* guarantees that, in the context of a single transaction, there is only one object instance that represents a particular database row. This avoids the previous problem and also allows for some caching to be done at the transaction level.
- *Process-scoped identity* goes one step further and guarantees that there is only one object instance representing the row in the whole process (JVM).

For a typical web or enterprise application, transaction-scoped identity is preferred.
Process-scoped identity offers some potential advantages in terms of cache utilization and the programming model for reuse of instances across multiple transactions; however, in a pervasively multithreaded application, the cost of always synchronizing shared access to persistent objects in the global identity map is too high a price to pay. It's simpler, and more scalable, to have each thread work with a distinct set of persistent instances in each transaction scope. Speaking loosely, we would say that Hibernate implements transaction-scoped identity. Actually, the Hibernate identity scope is the `Session` instance, so identical objects are guaranteed if the same persistence manager (the `Session`) is used for several operations. But a `Session` isn't the same as a (database) transaction—it's a much more flexible element. We'll explore the differences and the consequences of this concept in the next chapter. Let's focus on the persistence lifecycle and identity scope again.
If you request two objects using the same database identifier value in the same `Session`, the result will be two references to the same in-memory object. The following code example demonstrates this behavior, with everal `load()` operations in two `Sessions`:

```
Session session1 = sessions.openSession();
Transaction tx1 = session1.beginTransaction();
// Load Category with identifier value "1234"
Object a = session1.load(Category.class, new Long(1234) );
Object b = session1.load(Category.class, new Long(1234) );
if ( a==b ) {
        System.out.println("a and b are identical.");
}
tx1.commit();
session1.close();
Session session2 = sessions.openSession();
Transaction tx2 = session2.beginTransaction();
Object b2 = session2.load(Category.class, new Long(1234) );
if ( a!=b2 ) {
        System.out.println("a and b2 are not identical.");
}
tx2.commit();
session2.close();
```

Object references `a` and `b` not only have the same database identity, they also have the same Java identity since they were loaded in the same `Session`. Once outside this boundary, however, Hibernate doesn't guarantee Java

identity, so `a` and `b2` aren't identical and the message is printed on the console. Of course, a test for database identity—`a.getId().equals ( b2.getId() )`—would still return true. To further complicate our discussion of identity scopes, we need to consider how the persistence layer handles a reference to an object outside its identity scope. For example, for a persistence layer with transaction-scoped identity such as Hibernate, is a reference to a detached object (that is, an instance persisted or loaded in a previous, completed session) tolerated?

4.1.5 Outside the identity scope

If an object reference leaves the scope of guaranteed identity, we call it a *reference to a detached object*. Why is this concept useful?

In web applications, you usually don't maintain a database transaction across a user interaction. Users take a long time to think about modifications, but for scalability reasons, you must keep database transactions short and release database resources as soon as possible. In this environment, it's useful to be able to reuse a reference to a detached instance. For example, you might want to send an object retrieved in one unit of work to the presentation tier and later reuse it in a second unit of work, after it's been modified by the user.

You don't usually wish to reattach the entire object graph in the second unit of of work; for performance (and other) reasons, it's important that reassociation of detached instances be selective. Hibernate supports *selective reassociation of detached instances*. This means the application can efficiently reattach a *subgraph* of a graph of detached objects with the current ("second") Hibernate `Session`. Once a detached object has been reattached to a new Hibernate persistence manager, it may be considered a persistent instance, and its state will be synchronized with the database at the end of the transaction (due to Hibernate's automatic dirty checking of persistent instances).

Reattachment might result in the creation of new rows in the database when a reference is created from a detached instance to a new transient instance. For example, a new `Bid` might have been added to a detached `Item` while it was on the presentation tier. Hibernate can detect that the `Bid` is new and must be inserted in the database. For this to work, Hibernate must be able to distinguish between a "new" transient instance and an "old" detached instance. Transient instances (such as the `Bid`) might need to be saved; detached instances (such as the `Item`) might need to be reattached (and later updated in the database). There are several ways to distinguish between transient and detached instances, but the nicest approach is to look at the value of the identifier property. Hibernate can examine the identifier of a transient or detached object on reattachment and treat the object (and the associated graph of objects) appropriately. We discuss this important issue further in section 4.3.4, "Distinguishing between transient and detached instances." If you want to take advantage of Hibernate's support for reassociation of detached instances in your own applications, you need to be aware of Hibernate's identity scope when designing your application—that is, the `Session` scope that guarantees identical instances. As soon as you leave that scope and have detached instances, another interesting concept comes into play.

Equality is an identity concept that you, as a class developer, control and that you can (and sometimes have to) use for classes that have detached instances. Java equality is defined by the implementation of the `equals()` and `hashCode()` methods in the persistent classes of the domain model.

4.1.6 Implementing equals() and hashCode()

The `equals()` method is called by application code or, more importantly, by the Java collections. A `Set` collection, for example, calls `equals()` on each object you put in the `Set`, to determine (and prevent) duplicate elements.

First let's consider the default implementation of `equals()`, defined by `java.lang.Object`, which uses a comparison by Java identity. Hibernate guarantees that there is a unique instance for each row of the database inside a `Session`. Therefore, the default identity `equals()` is appropriate if you never mix instances—that is, if you never put detached instances from different sessions into the same `Set`. (Actually, the issue we're exploring is also visible if detached instances are from the same session but have been serialized and deserialized in different scopes.) As soon as you have instances from multiple sessions, however, it becomes possible to have a `Set` containing two `Item`s that each represent the same row of the database table but don't have the same Java identity. This would almost always be semantically wrong. Nevertheless, it's possible to build a complex application with identity (default) equals as long as you exercise discipline when dealing with detached objects from different sessions (and keep an eye on serialization and deserialization). One nice thing about this approach is that you don't have to write extra code to implement your own notion of equality.

However, if this concept of equality isn't what you want, you have to override `equals()` in your persistent classes. Keep in mind that when you override `equals()`, you always need to also override `hashCode()` so the

two methods are *consistent* (if two objects are equal, they must have the same hashcode). Let's look at some of the ways you can override `equals()` and `hashCode()` in persistent classes.

Using database identifier equality

A clever approach is to implement `equals()` to compare just the database identifier property (usually a surrogate primary key) value:

```
public class User {
...
public boolean equals(Object other) {
        if (this==other) return true;
                if (id==null) return false;
                        if ( !(other instanceof User) ) return false;
                        final User that = (User) other;
                        return this.id.equals( that.getId() );
}
public int hashCode() {
        return id==null ?
        System.identityHashCode(this) :
        id.hashCode();
        }
}
```

Notice how this `equals()` method falls back to Java identity for transient instances (if `id==null`) that don't have a database identifier value assigned yet. This is reasonable, since they can't have the same persistent identity as another instance.

Unfortunately, this solution has one huge problem: Hibernate doesn't assign identifier values until an entity is saved. So, if the object is added to a `Set` before being saved, its hash code changes while it's contained by the `Set`, contrary to the contract of `java.util.Set`. In particular, this problem makes cascade save (discussed later in this chapter) useless for sets. We strongly discourage this solution (database identifier equality).

Comparing by value

A better way is to include all persistent properties of the persistent class, apart from any database identifier property, in the `equals()` comparison. This is how most people perceive the meaning of `equals()`; we call it *by value* equality. When we say "all properties," we don't mean to include collections. Collection state is associated with a different table, so it seems wrong to include it. More important, you don't want to force the entire object graph to be retrieved just to perform `equals()`. In the case of `User`, this means you shouldn't include the `items` collection (the items sold by this user) in the comparison. So, this is the implementation you could use:

```
public class User {
...
public boolean equals(Object other) {
if (this==other) return true;
if ( !(other instanceof User) ) return false;
final User that = (User) other;
if ( !this.getUsername().equals( that.getUsername() )
return false;
if ( !this.getPassword().equals( that.getPassword() )
return false;
return true;
}
public int hashCode() {
int result = 14;
result = 29 * result + getUsername().hashCode();
result = 29 * result + getPassword().hashCode();
return result;
}
}
```

However, there are again two problems with this approach:

- Instances from different sessions are no longer equal if one is modified (for example, if the user changes his password).
- Instances with different database identity (instances that represent different rows of the database table) could be considered equal, unless there is some combination of properties that are guaranteed to be unique (the database columns have a unique constraint). In the case of `User`, there is a unique property: `username`.

To get to the solution we recommend, you need to understand the notion of a *business key*.

Using business key equality

A *business key* is a property, or some combination of properties, that is unique for each instance with the same database identity. Essentially, it's the natural key you'd use if you weren't using a surrogate key. Unlike a natural primary key, it isn't an absolute requirement that the business key never change—as long as it changes rarely, that's enough.

We argue that every entity should have a business key, even if it includes all properties of the class (this would be appropriate for some immutable classes). The business key is what the user thinks of as uniquely identifying a particular record, whereas the surrogate key is what the application and database use. *Business key equality* means that the `equals()` method compares only the properties that form the business key. This is a perfect solution that avoids all the problems described earlier. The only downside is that it requires extra thought to identify the correct business key in the first place. But this effort is required anyway; it's important to identify any unique keys if you want your database to help ensure data integrity via constraint checking. For the `User` class, `username` is a great candidate business key. It's never null, it unique, and it changes rarely (if ever):

```
public class User {
...
public boolean equals(Object other) {
if (this==other) return true;
if ( !(other instanceof User) ) return false;
final User that = (User) other;
return this.username.equals( that.getUsername() );
}
public int hashCode() {
return username.hashCode();
}
}
```

For some other classes, the business key might be more complex, consisting of a combination of properties. For example, candidate business keys for the `Bid` class are the item ID together with the bid amount, or the item ID together with the date and time of the bid. A good business key for the `BillingDetails` abstract class is the `number` together with the type (subclass) of billing details. Notice that it's almost never correct to override `equals()` on a subclass and include another property in the comparison. It's tricky to satisfy the requirements that equality be both symmetric and transitive in this case; and, more important, the business key wouldn't correspond to any well-defined candidate natural key in the database (subclass properties may be mapped to a different table).

You might have noticed that the `equals()` and `hashCode()` methods always access the properties of the other object via the getter methods. This is important, since the object instance passed as `other` might be a proxy object, not the actual instance that holds the persistent state. This is one point where Hibernate isn't completely transparent, but it's a good practice to use accessor methods instead of direct instance variable access anyway. Finally, take care when modifying the value of the business key properties; don't change the value while the domain object is in a set.