CHAPTER 7

Understanding Entity Beans

AN entity bean is a component that encapsulates the implementation of a business entity or a business process. The encapsulation of a business entity or process is typically independent of the client application that uses the entity bean. As a result, an entity bean can be reused by many client applications.

This chapter focuses on the basics of entity beans from a programming point of view. The chapter describes the client view of an entity bean—the view of the application developer who uses an entity bean—and the view of the developer of the entity bean itself. We also describe the life cycle of entity bean class instances and show how the container manages the instances at runtime.

Clients of entity beans, similar to session bean clients, use the methods of the home and component interfaces. However, because the life cycles of an entity object and a session object differ, an entity bean client takes a different approach to using the home interface methods for creating and removing entity objects. In addition, each entity object has a unique identity, which allows the home interface to define find methods for locating entity bean instances.

A bean developer must write the implementation of the entity bean's business logic in addition to the life-cycle-related methods. The bean developer is also concerned about entity object persistence. The state of an entity object is stored in a resource manager, such as a database or other persistent storage. The entity bean methods access the state in the resource manager, using either container-managed persistence (CMP) or bean-managed persistence (BMP).

This chapter also explains how the container manages entity bean instances. Although not all developers need to know this information, it is interesting to see what happens beneath the surface during the various method invocations, passivation and activation, and transactions.

7.1 Client View of an Entity Bean

We begin by describing the client view of an entity bean: the view seen by an application developer who uses an entity bean in a client application. Note that the client application developer is typically different from the developer—or company—that developed the bean.

An entity bean is a component that gives its client a true object-oriented abstraction of a business entity or a business process. For example, a real-life business entity may be an account, employee, customer, and so forth. A business process, on the other hand, can be the sequence involved in granting a loan approval, opening a bank account, scheduling a meeting, and so forth. When an entity bean is used to implement a business entity or a process, each individual business entity or process is represented by an entity object.

In most cases, entity beans are used with a local client view. Such entity beans are developed with local interfaces, which enable the bean to take advantage of the complete range of the EJB 2.0 and 2.1 architectural features. For example, an entity bean implemented with container-managed persistence can participate in container-managed relationships with other entity beans. With container-managed relationships, the EJB container manages the persistent relationships between entity beans much like it manages a bean's persistence.

Entity beans with local interfaces make the most use of the resources of the EJB environment. However, the beans are restricted to being colocated on the same JVM as their clients. When an entity bean's application is such that the bean must provide a remote client view, the developer can take steps to use the local-view advantages and still retain distributed capabilities. Principally, the developer can provide a session bean with a remote client view as a facade to entity beans with local views. Or, the developer can provide the entity bean itself with a remote interface in addition to its local interface.

Four concepts define the client view of an entity bean:

1. Home interface
2. Component interface: local and remote interfaces
3. Primary key and object identity
4. Life cycle of an entity object

In the following sections, we explain and illustrate how a client uses the home and component interfaces to manipulate entity objects. We also discuss the role of the primary key in the client view and explain the life cycle of an entity bean.

7.1.1 Home Interface

A client uses the home interface to manage the life cycle of individual entity objects. Life-cycle operations involve creating, finding, and removing entity objects. Specifically, the client uses the home interface methods to create new entity objects and to find and remove existing ones.

A client can also use business methods on the home interface to perform aggregate operations on entity objects. Keep in mind that these aggregate operations are not specific to any particular entity object.

In addition, a client can use the home interface to obtain the `javax.ejb.EJBMetaData` interface for the entity bean and to obtain a handle for the home interface. These functions apply only to entity beans implementing a remote home interface.

Although the signatures of the create and find methods may be different for each entity bean, the life-cycle operations are uniform across all entity beans. This uniformity makes it easier for a client developer to use entity beans supplied by other developers.

Let's look at the example `AccountHome` interface, which was used in Chapter 2. Code Example 7.1 shows the `AccountHome` interface with a local client view:

```
import javax.ejb.CreateException;
import javax.ejb.FinderException;
public interface AccountHome extends javax.ejb.EJBLocalHome {
   // create methods
   Account create(String lastName, String firstName)
      throws CreateException, BadNameException;
   Account createBusinessAcct(String businessName)
      throws CreateException;
   ...

   // find methods
   Account findByPrimaryKey(AccountKey primaryKey)
      throws FinderException;
   Collection findInactive(Date sinceWhen)
      throws FinderException, BadDateException;
```

```
    ...
    // home methods
    public void debitAcctFee(float fee_amt)
        throws OutofRangeException;
    ...
}
```

Code Example 7.1 `AccountHome` Interface

Recall that the `AccountHome` interface previously implemented a remote client view and thus extended the `javax.ejb.EJBHome` interface. A home interface for an entity bean that implements a local client view extends the `javax.ejb.EJBLocalHome` interface. See Section 2.3.2, Enterprise Bean Home Interfaces, on page 33. The next sections explain the client's use of the home interface.

Locating the Home Interface

The client must first obtain the home interface to use it. A client obtains the local home interface for the AccountEJB entity bean from the client's environment, using the JNDI API, casting the returned value to the home interface type. The deployer has configured the home interface in the JNDI namespace (Code Example 7.2):

```
Context initCtx = new InitialContext();
AccountHome accountHome = (AccountHome) initCtx.lookup(
    "java:comp/env/ejb/CheckingAccountEJB");
```

Code Example 7.2 Obtaining a Local Home Interface

If the AccountEJB entity bean uses a remote client view, a client also obtains the remote home interface for the bean from the client's environment, using JNDI. However, the client must use the `PortableRemoteObject.narrow` method to cast the value returned from JNDI to the home interface type (Code Example 7.3):

```
Context initCtx = new InitialContext();
AccountHome accountHome = (AccountHome)PortableRemoteObject.narrow(
```

```
initCtx.lookup("java:comp/env/ejb/CheckingAccountEJB"),
AccountHome.class);
```

Code Example 7.3 Obtaining a Home Interface for a Bean with a Remote View

Using Create Methods

An entity bean home interface defines zero or more create methods, each representing a different way to create a new entity object. Each create method must begin with the word `create` and may be followed by a descriptive word. The number and types of the input parameters of the create methods are entity bean specific. The return value type for all create methods is always the entity bean's component interface.

Returning to our example, the client can use the `AccountHome` interface to create new `Account` objects, as follows:

```
Account account1 = accountHome.create("Matena", "Vlada");
Account account2 = accountHome.create("Stearns", "Beth");
Account businessAcct = accountHome.createBusinessAcct(
       "ComputerEase Publishing");
```

It is important to understand that when the client invokes a create operation, the entity bean creates the representation of the entity object's state in a persistent store, such as a relational database. This is in contrast to invoking a create method on a session bean. Invoking a create method on a session bean results only in the creation of a session bean instance; it does not result in the creation of persistent state in a persistent store.

It is possible for a home interface to define no create methods, and sometimes this approach is useful and preferable. Because the entity object's state exists in the resource manager independently from the entity bean and its container, it is possible to create or remove an entity object directly in the resource manager. (The section Entity Object State and Persistence on page 176 explains how a resource manager manages the state of an entity object.)

An application is not limited to going through the entity bean and its container to create or remove the object. For example, an application can use SQL statements to create or remove an `Account` object in a relational database. In some situations, the bean developer does not want to allow the entity bean clients to create entity objects and instead wants to ensure that the entity objects are created solely

by other means. In such a case, the entity bean's home interface would have zero create methods. This is discussed in more detail in Section 7.2.7, Using Entity Beans with Preexisting Data, on page 235.

Using Find Methods

The home interface defines one or more find methods. A client uses the find methods to look up entity objects that meet given criteria.

All entity bean home interfaces define a `findByPrimaryKey` method to allow the client to find an entity object by its primary key. The `findByPrimaryKey` method takes a single input parameter with a type that is the entity bean's primary-key type. The return value type is the entity bean's component interface.

Our example AccountEJB client may use the `findByPrimaryKey` method, as follows:

```
AccountKey pk = new AccountKey();
pk.setAccountNumber("100-300-423");

try {
    Account account = accountHome.findByPrimaryKey(pk);
} catch (ObjectNotFoundException ex) {
    // account with the given primary key does not exist
}
```

If the `findByPrimaryKey` method completes successfully, the client knows that the entity object with the given primary key exists. If the entity object does not exist when the client invokes the `findByPrimaryKey` method, the method throws `javax.ejb.ObjectNotFoundException` to the client.

In addition to the required `findByPrimaryKey` method, the home interface may include additional find methods. The number and types of the input arguments of these additional find methods are entity bean specific. A find method's return value type is either the entity bean's component interface or `java.util.Collection`. Any find methods that can potentially find more than one entity object should define the return value type as `java.util.Collection`; those that can return at most one entity object should define the return value type as the component interface type.

The following example illustrates using a find method, `findInactive`, that may return more than one entity object:

```
Date sinceDate = ...;
Collection inactiveAccounts = accountHome.findInactive(sinceDate);
Iterator it = inactiveAccounts.iterator();
while (it.hasNext()) {
    Account acct = (Account) it.next(),
    // do something with acct
    acct.debit(100.00);
}
```

The `findInactive` method returns a collection of accounts that have been inactive since a given date. The client uses an iterator to obtain the individual entity objects returned in `Collection`.

Note that, had the bean used a remote client view, the client must use the `PortableRemoteObject.narrow` operation to cast an object retrieved from the collection to the entity bean's remote interface type. The client cannot simply retrieve the next object in `Collection`:

```
Account acct = (Account) it.next();
```

Instead, the client performs the retrieval as follows:

```
Account acct = (Account)PortableRemoteObject.narrow(it.next(),
                  Account.class);
```

Using `remove` Methods

As noted earlier, all entity bean local home interfaces extend the `javax.ejb.EJBLocalHome` interface, and all entity bean remote home interfaces extend the `javax.ejb.EJBHome` interface. The `EJBLocalHome` interface defines one `remove` method, whereas the `EJBHome` interface defines two `remove` methods. A client uses the `EJBHome` or `EJBLocalHome` `remove` methods to remove the entity objects specified by the method's input parameter.

The `void remove(Object primaryKey)` method is defined by the `EJBHome` and `EJBLocalHome` interfaces. This method is used to remove an entity object by a given primary key. The `void remove(Handle handle)` method, defined by the `EJBHome` interface only for remote home interfaces, is used to remove an entity object identified by a given handle. The following code illustrates using a `remove` method:

```
AccountKey pk = new AccountKey();
pk.setAccountNumber("100-300-423");
accountHome.remove(pk);
```

It is important to understand that successful execution of a `remove` method results in the removal of the representation of the entity object's state from the resource manager that stores the state. In the previous example, the `remove` method results in the removal of the specified account record from the database.

Using Home Business Methods

The home interface may also define business methods that operate across all bean instances. Suppose that a client wants to periodically subtract a set fee from all account records. Rather than having to write code to retrieve all `Account` instances and subtract a set amount from each account's balance, a client might use `AccountHome`'s `debitAcctFee` method to accomplish this same operation across all `Account` instances. For example:

```
if (monthlyUpdate) {
     accountHome.debitAcctFee(acctFee);
}
```

7.1.2 Component Interface

To get a reference to an existing entity object's component interface—either its remote interface or its local interface—the client can

- Receive the reference as a result of a create method
- Find the entity object by using a find method defined in the entity bean's home interface
- Receive the reference as an input parameter or a method result in a method call

Once it obtains the reference to the entity bean's component interface, the client can do a number of things with that reference:

- Invoke business methods on the entity object through the component interface
- Obtain a reference to the entity bean's home interface

- Pass the reference in a parameter or as a result of a remote or local method call
- Obtain the entity object's primary key
- Remove the entity object

Code Example 7.4 shows the definition of the `Account` component interface, which is a local interface:

```
public interface Account extends javax.ejb.EJBLocalObject {
    double getBalance();
    void credit(double amount);
    void debit(double amount) throws InsufficientFundsException;
    ...
}
```

Code Example 7.4 `Account` Interface

Had the AccountEJB bean implemented a remote client view, each of the `Account` component interface methods would throw `java.rmi.RemoteException` in addition to its other exceptions.

Code Example 7.5 illustrates using the component interface for a bean with a local client view:

```
// Somehow obtain a reference to an Account object.
Account acct = ...;

// Invoke business methods.
double balance = acct.getBalance();
acct.debit(200.00);
acct.credit(300.00);

// Obtain the primary key.
AccountKey pk = (AccountKey)acct.getPrimaryKey();

// Obtain the home interface.
AccountHome home = (AccountHome) acct.getEJBHome();

// Pass the entity object as a parameter in a method call.
```

```
// (Foo is an enterprise bean's local home or component interface.)
Foo foo = ...;    // Foo
foo.someMethod(acct, ...);

// Remove the entity object.
acct.remove();
```

Code Example 7.5 Using the Component Interface

7.1.3 Primary Key and Object Identity

Every entity object has an identity unique within the scope of its home interface. The primary key determines this identity. If two entity objects with the same home interface have the same primary key, they are considered identical entity objects. If they have a different primary key, they are considered different entity objects.

A client can test whether two entity object references refer to the same entity object by using the `isIdentical` method. The following code segment illustrates using the `isIdentical` method to compare two object references to determine whether they are identical:

```
Account acct1 = ...;
Account acct2 = ...;

if (acct1.isIdentical(acct2)) {
      // acct1 and acct2 refer to the same entity object
} else {
      // acct1 and acct2 refer to different entity objects
}
```

Alternatively, if it obtains two entity object references from the same home interface, the client can determine if these objects are identical by comparing their primary keys:

```
AccountHome accountHome = ...;
Account acct1 = accountHome.findOneWay(...);
Account acct2 = accountHome.findAnotherWay(...);

if (acct1.getPrimaryKey().equals(acct2.getPrimaryKey())) {
      // acct1 and acct2 refer to the same entity object
```

```
} else {
      // acct1 and acct2 refer to different entity objects
}
```

Note that comparing primary keys is valid only when comparing objects obtained from the same home interface. When objects are obtained from different home interfaces, the client must use the `isIdentical` method on one of the objects to perform the comparison.

7.1.4 Entity Object Life Cycle

Figure 7.1 illustrates an entity object's life cycle, as seen from the perspective of a client. In the diagram, the term *referenced* means that the client holds an object reference for the entity object. The figure illustrates the following points about an entity object's life cycle:

- **Creating an entity object**
  - A client can create an entity object by invoking a create method defined in the entity bean's home interface.
  - It is possible to create an entity object without involving the entity bean or its container. For example, using a direct database insert can create the representation of the entity object's state in the resource manager—that is, in the database.
- **Finding an entity object**—A client can look up an existing entity object by using a find method defined in the home interface.
- **Invoking business methods**—A client that has an object reference for the entity object can invoke business methods on the entity object.
- **Understanding the life cycle**—The life cycle of the entity object is independent of the life cycle of the client-held object references. This means that an entity object is not removed when it is no longer referenced by a client. Likewise, the existence of an object reference does not ensure the existence of the entity object.

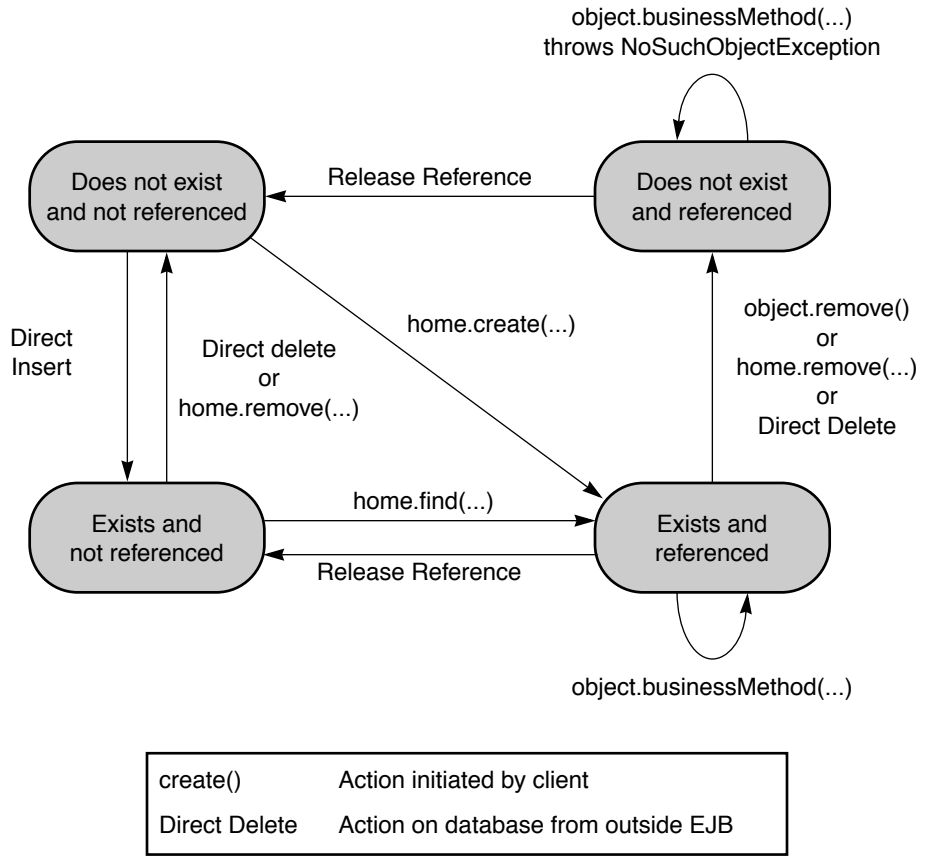

Figure 7.1 Client View of an Entity Object Life Cycle

- **Removing an entity object**
  - A client can remove an entity object by using one of the `remove` methods defined in the home and component interfaces. If a client attempts to invoke a business method on an entity object after the object has been removed, the client receives `NoSuchObjectException`.
  - It is possible to remove an entity object without involving the entity bean or its container. For example, using a direct database delete can remove the representation of the entity object's state from the resource manager: the database. If a client attempts to invoke a business method on an entity object after its state has been removed from the database, the client receives `NoSuchObjectLocalException` (or `NoSuchObjectException` if a remote client).

Entity objects are considered to be, in general, persistent objects. An entity object can be accessed concurrently through multiple JVMs. The lifetime of an entity object is not limited by the lifetime of the JVM process in which the entity bean instances execute.

Although the crash of the JVM may result in the rollback of current transactions, it does not destroy previously created entity objects; nor does it invalidate the references to the component and home interfaces held by clients.

An entity object remains accessible to its clients as long as the representation of its state is maintained in the resource manager or until a reconfiguration of the bean or container invalidates the object references and handles held by the clients. This can happen, for example, when the entity bean is uninstalled from the container or if the container is reconfigured to listen on a different network address.

Multiple clients can access the same entity object concurrently. If so, the container uses transactions to isolate the clients' work from one another. This is explained in Section 7.2.6, Concurrent Invocation of an Entity Object, on page 233.

7.2 Bean Developer View of an Entity Bean

The bean developer's view of an entity bean differs from that of the client. Essentially, the bean developer is responsible for the implementation of the methods defined in the bean's component and home interfaces, as well as the callback methods of the `EntityBean` interface. The developer needs to know how to implement correctly the methods defined by the home interface—the find, create, and home methods—the business methods, and the methods defined by the `EntityBean` interface. These method implementations access the state of the entity objects maintained in a resource manager. As a result, the bean developer needs to understand entity object state and persistence to implement the entity bean methods optimally.

We begin this section by describing entity object state and persistence. The entity object state is managed by a resource manager. Management of state is separate from the container's management of entity bean instances. The developer can use either the CMP or the BMP approach to access object state. Each of these two persistence approaches has advantages and drawbacks.

For entity beans using CMP, the developer can use the EJB QL query language to implement the find methods. EJB QL is an SQL-like query language. The developer can also define additional queries for use internal to the bean itself, using `ejbSelect` methods. We discuss these concepts in this section.

This section also describes various approaches to using the `ejbLoad` and `ejbStore` methods. The container invokes these methods on the bean implementation. The developer can use these methods to take advantage of the container's cache management capabilities and to maximize performance. (See Section 7.2.4, Caching Entity Bean State, on page 219.) This section ends with a discussion on how the container manages multiple client invocations on an entity object.

7.2.1 Entity Object State and Persistence

The methods of an entity bean class access the object's state in a resource manager. Most resource managers, such as relational databases, manage entity objects' state externally to the bean instances (Figure 7.2).

Separating the state of the entity objects from the instances of the entity bean class has the following advantages:

- **Facilitates persistence and transactions**
  - Separating an entity object's state from the bean class instance allows the entity object's state to be persistent. Separation permits the life cycle of the entity object's state to be independent of the life cycle of the entity bean class instances and of the life cycle of the JVMs in which the instances are created.
  - The resource manager, instead of the entity bean or the EJB container, can handle the implementation of the ACID properties—atomicity, consistency, isolation, and durability—which pertain to transactions.
- **Promotes the implementation of the EJB server**
  - Separation makes it possible to implement the entity object's state to be accessible concurrently from multiple JVMs, even when the JVMs run on different network nodes. This is essential to a high-end implementation of an EJB server.

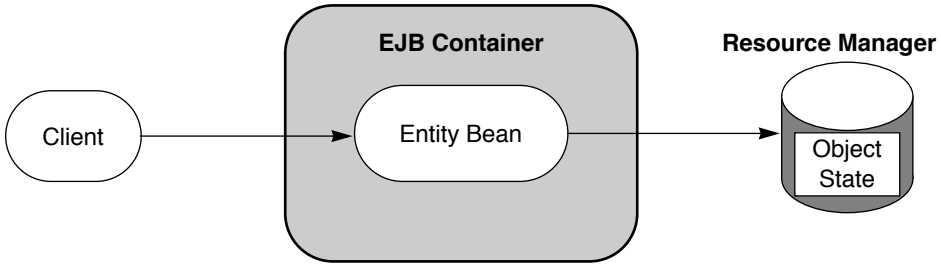

Figure 7.2 Entity Object's State Managed by Resource Manager

  - It makes it possible to implement a highly available EJB server. Should a node of the JVM be unavailable, another JVM can access the entity object's state.

- **Improves accessibility between Java and non-Java applications**
  - It makes it possible to externalize an entity object's state in a representation suitable for non-Java applications. For example, if a relational database keeps the state of entity objects, the state is available to any application that can access the database via SQL statements.
  - It makes it possible to present data residing in an enterprise's databases as entity beans. Similarly, it allows the creation of an entity bean facade on top of the client interfaces of the enterprise's non-Java application. This client-view entity bean of the enterprise's preexisting data or applications makes it easier to develop and integrate new Java applications with the legacy data or applications.

The entity bean architecture is flexible about the choice of the type of resource manager in which to store an entity object's state. Examples of resource managers in which the state of an entity bean can be stored are

- A relational database system
- An application system, such as an ERP system or mainframe application
- A nonrelational database, such as a hierarchical database or an object-oriented database
- Some form of fast secondary-memory resource manager that may be provided by the EJB container as an option

The state of most entity beans is typically stored in a resource manager external to the EJB container—these are the database or application systems noted in the first three examples just presented. The fast secondary memory integrated with the EJB container is a special example. If provided, it typically is used only for storing the state of short-lived entity objects with states that are not accessed by other applications running outside the EJB container.

Because a resource manager maintains the state of an entity object, an entity bean instance must use an API to access the state of the associated entity object. (Associating an instance with an entity object is called *object activation,* and it is explained in Section 7.2.3, Entity Bean Instance Life Cycle, on page 196.) An

entity bean instance can access the state of its associated entity object by using two access styles: BMP and CMP. Because CMP is the preferred approach, it is discussed first.

Container-Managed Persistence

Container-managed persistence (CMP) is a resource manager–independent data access API tailored for use by entity beans. CMP offers distinct advantages over bean-managed persistence (BMP). Probably the main advantage of CMP involves database access code. It is simpler to write database access code with CMP than to write such code with BMP. Database access with BMP usually entails coding JDBC calls. Using CMP, the developer need only describe *what* state is to be stored persistently; the container takes care of *how* the persistence is performed. With BMP, on the other hand, the developer is responsible for both what state to persist and how that state is to be persisted.

Another important advantage of CMP is that it greatly enhances an entity bean's portability. A developer uses the same API—that is, the EJB CMP methods—regardless of the underlying type of resource manager. The same entity bean can thus be used with any type of resource manager.

CMP enables the development of entity beans with implementations that can be used with multiple resource manager types and database schemas. The CMP architecture allows ISVs to develop entity beans that can be adapted at deployment to work with customers' preexisting data. Because different customers use different resource manager types and have different database schemas, the deployer needs to adapt an entity bean to each customer's resource manager type and its database schema. Most important, the entity bean developer does not have to write a different implementation of the bean for each customer.

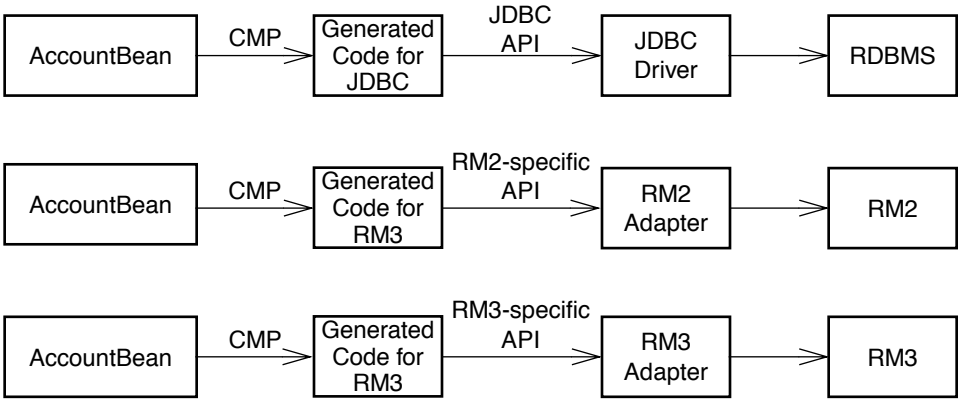

Figure 7.3 AccountBean Entity Bean Using CMP to Access Resource Managers

Figure 7.3 shows how an entity bean using CMP uses the various resource manager–specific APIs to access data in the respective resource managers. Note that the developer has to write only one entity bean class, regardless of the resource manager type. Based on the mapping done by the deployer, the container generates the appropriate data access code for the resource manager API. Therefore, the developer does not need to know or care about the actual resource manager–specific code.

The performance of CMP entity beans is also likely to be significantly better than that of BMP entity beans or session beans using JDBC. This may sound counterintuitive, because developers assume they can code database access calls in a way that is best for their application. However, obtaining good database access performance requires more than correctly coding the access calls. Rather, obtaining good performance requires such techniques as advanced knowledge of database-specific features, connection pooling, an optimized caching strategy for reducing the number of database round-trips, and more. System-level experts who build containers, persistence managers, and application servers know these advanced techniques the best. With CMP, because the container has full control over persistence management, it can optimally manage database access to achieve the highest performance. This fact has been observed with EJB-based benchmarks on several application server products.

The EJB 2.0 specification significantly extended the functionality of CMP. The persistent state of an entity bean is described by JavaBeans-like properties, which are represented in the bean class as get and set accessor methods. For example, a persistent field `foo` would be represented by the `getFoo` and `setFoo` methods in the bean class. These get and set methods allow code in the bean class to access the data items comprising the entity object's state. CMP fields are virtual fields and do not appear as instance variables in the entity bean's implementation class. The implementation class instead defines get and set methods, which are declared `public` and `abstract`, for each such `public` variable. The bean provides no implementation for these methods. Instead, the EJB container provides the method implementations. The implementation class, too, is an abstract class for entity beans that use CMP.

For example, the same `AccountBean` class coded using EJB 2.1 CMP would look as shown in Code Example 7.6:

```
public abstract class AccountBean implements EntityBean {
   // Accessor methods for container-managed fields
```

```
    public abstract String getAccountNumber();    // accountNumber
    public abstract void setAccountNumber(String v);

    public abstract double getBalance();         // balance
    public abstract void setBalance(double v);
    ...
    // Business methods
    public void debit(double amount) {
       setBalance(getBalance() - amount);
    }
}
```

Code Example 7.6 AccountBean Entity Bean Using EJB 2.1 CMP

Note also that entity beans using CMP can implement a local or a remote client view. However, an entity bean is not required to have a local client view to use CMP. Because the local interface exposes a bean's methods only to other beans that reside within the same container, there can be fine-grained access between clients and entity beans without an undue performance overhead. Entity beans with local interfaces not only can expose their state to clients but also can use pass-by-reference semantics to pass state between related bean instances.

Applications migrating from an EJB 1.0 or 1.1 approach to the EJB 2.0 or 2.1 specification should keep in mind the change to parameter passing. The remote interface approach uses pass-by-value semantics, whereas pass-by-reference mode with local interfaces may result in unintended changes to the state of a passed value or object. With pass-by-value semantics, the invoked method's actions affect only its local copy of a value or object. With pass-by-reference semantics, there is only one copy of a particular value or object. All involved methods reference and act on the state of that one value or object.

In the EJB 1.0 and 1.1 specifications, the CMP API depended on mapping the instance variables of an entity bean class to the data items representing an entity object's state in the resource manager. For example, Code Example 7.7 shows the same `AccountBean` class coded using CMP:

```
public class AccountBean implements EntityBean {
   // container-managed fields
   public String accountNumber;
   public double balance;
```

```
    ...
    public void debit(double amount) {
       balance = balance - amount;
    }
}
```

Code Example 7.7 AccountBean Entity Bean Using CMP

At deployment, the deployer uses tools to generate code that implements the mapping of the instance fields to the data items in the resource manager. Although the EJB 1.1 field-based approach to CMP is still supported for applications implemented prior to the EJB 2.0 specification, this book focuses on the EJB 2.0 CMP approach.

Container-Managed Relationships

Along with an enhanced CMP approach, the EJB 2.1 specification supports container-managed relationships (CMRs) as part of the CMP architecture. Multiple entity beans are now allowed to have relationships among themselves, referred to as CMRs.

Container-managed relationships are described by CMR fields in the deployment descriptor. The EJB container supports one-to-one, one-to-many, and many-to-many relationships, automatically managing the relationships and maintaining their referential integrity. For example, consider a many-to-many bidirectional relationship, such as one between a Supplier entity bean and a Parts entity bean. If a `Supplier` instance stops providing a particular part, the container automatically removes that `Supplier` from the set of available suppliers for that `Parts` instance.

CMR fields are handled in the same way as CMP fields. CMR fields do not appear as instance variables in the entity bean's implementation. Instead, the implementation defines `public`, `abstract` get and set methods for each CMR `public` variable. The EJB container provides the method implementations, rather than the bean itself. An entity bean must provide a local client view so that it can be used as the target of a container-managed relationship.

For one-to-one and many-to-one relationships, the get and set accessors for a CMR field are defined using the related bean's local interface type. For one-to-many and many-to-many relationships, the get and set accessors use the Java `Collection` interface, as there are many instances of the related bean.

The bean provider specifies the relationships between entity beans in the deployment descriptor. At deployment, these relationship specifications become the schema definition, and the relationships may be captured in a relational database or other resource manager. For example, a relationship between two beans may appear as a foreign key relationship in a relational database.

Container-managed relationships are also key to the powerful querying features of the EJB QL query language. They allow a developer to concisely write a query that operates over several related beans.

EJB QL Query Language

The EJB 2.0 specification introduced EJB Query Language, or EJB QL, a query language that is intended to be portable across EJB containers. EJB QL is an SQL-like language for expressing queries over entity beans with container-managed persistence. EJB QL defines queries for find and select methods for entity beans with container-managed persistence.

Prior to the EJB 2.1 specification, the manner and language for forming and expressing queries for find methods were left to each individual application server. Many application server vendors let developers form queries using SQL. However, other vendors used their own proprietary language specific to their particular application server product.

In addition, each application server provided its own tool for expressing a query. Because the developer must know the database table and column names to correctly form an SQL query, application server tools that rely on SQL provided a mapping between the database names and SQL statements to help the developer.

Under the previous versions of the EJB architecture, each application server handled queries in its own manner. Migrating from one application server to another often required you to use the new server's tool to rewrite or reform your existing queries. For example, you might have to use the new server's tool to rewrite the SQL for the find methods used by the application's entity beans.

Application servers developed with early versions of the EJB architecture may also support different physical representations of the query. For example, the server may store the query in the deployment descriptor, or it may store the query someplace else. If kept in the deployment descriptor, the query may be identified with a tag unique to that particular server. These factors combined to reduce the portability of your enterprise bean application among various application servers.

EJB QL corrects many of these inconsistencies and shortcomings. EJB QL, designed to be an SQL-like language, defines find and select query methods spe-

cifically for entity beans with container-managed persistence. EJB QL's principal advantage over SQL is its portability across EJB containers and its ability to navigate entity bean relationships.

EJB QL allows querying or navigation over entity bean relationships. A query can begin with one entity bean and from there navigate to related beans. For example, the query can start with an Order bean and then navigate to the Order's line items. The query can also navigate to the products referenced by the Order's individual line items. However, this navigation requires that the bean relationships be expressed as container-managed relationships in the deployment descriptor.

With EJB QL, you can specify the query during in the application development process, and that query is kept in the deployment descriptor. Therefore, you have two options for writing the query: You can use the application server's query-related tools, or you can write the query by manually editing the deployment descriptor. The container generates the query language implementation for queries specified in the deployment descriptor.

For example, suppose that CustomerBean had a one-to-many relationship with AccountBean through the container-managed relationship field `accounts`. You want to form an EJB QL query to traverse from CustomerBean to AccountBean and return all customers with large accounts. You might write the query as follows:

```
SELECT DISTINCT OBJECT(cust)
FROM Customer cust, IN(cust.accounts) acct
WHERE acct.balance > ?1
ORDER BY cust.firstName
```

In CustomerBean's home interface, a developer might define the find method associated with this query as follows:

```
public Collection findLargeCustomers(int minimumBalance)
    throws FinderException;
```

As you can see, the EJB QL query looks similar to an SQL query: The query has a `SELECT` clause, a `FROM` clause, an optional `WHERE` clause, and an optional `ORDER BY` clause.

The EJB QL query names beans using their abstract schema names. A bean's abstract schema name, which is declared in its deployment descriptor, uniquely

identifies the bean in an EJB QL query. In the preceeding query, the abstract schema name of the CustomerBean is `Customer`.

The `SELECT` clause specifies the return type of the query. If the return values are to be unique—that is, no duplicates—add the `DISTINCT` keyword to the clause. You can indicate the return type by using an identification variable with the syntax `OBJECT(cust)`. This specifies that the query returns objects that are values of the identification variable `cust`. The return type may also be indicated by a path expression of the form `accounts.customer`, as long as the path expression ends in a single-valued (one-to-one or many-to-one) CMR or CMP field. You might use a path expression, for example, to implement a find method, `findLargeAccounts`, that returns a `Collection` of accounts. A path expression is the construct used to navigate from one bean to a second bean, using a CMR field of the first bean.

The return value for find methods can be only the type of the bean on whose home interface the find method is defined. There are no such restrictions on the return value for select methods defined in a bean class; their return values may be Java primitive types or bean types, or they may be collections of these types.

Similar to SQL, the return type of an EJB QL query may also be the result of an aggregate function: `AVG`, `MAX`, `MIN`, `SUM`, or `COUNT`. For example, `SELECT COUNT(cust)` returns a count of all `Customer` objects in the result of the query. The return type of the `ejbSelect` method whose query uses the `COUNT` function needs to be the Java primitive type `long`. The return type of an `ejbSelect` method using an `AVG` function needs to be `double`.

The `FROM` clause declares identification variables used in the `SELECT` and `WHERE` clauses. There are two ways to declare identification variables:

1. Declare an identification variable to be of a particular bean type—for example, `Customer cust`.
2. Use a path expression that ends in a collection-valued CMR field—that is, a one-to-many or many-to-many relationship, such as `IN(cust.accounts) acct`.

The `FROM` clause in EJB QL is similar in behavior to the `FROM` clause in SQL. With both `FROM` clauses, the effect of declaring multiple identification variables is a Cartesian product of the values of each of the identification variables. This means that if any of the clause's identification variables refers to a bean that has zero instances, the return values of the query will be empty.

The `WHERE` clause defines conditions that restrict the set of values returned from the query. The `WHERE` clause can use operators similar to SQL, including

arithmetic (+, -, *, /), comparison (=, >, <, >=, <=, <>), and logical (`NOT`, `AND`, `OR`) operators. Like SQL, EJB QL defines additional constructs for use in the `WHERE` clause, including

- **`BETWEEN`**—For example, `WHERE acct.balance BETWEEN 10000 AND 20000`
- **`LIKE`**—For example, `WHERE cust.name LIKE 'a%'`
- **`NULL, EMPTY` comparisons**—For example, `WHERE cust.name IS NULL AND cust.accounts IS EMPTY`
- **`IN` comparisons**—For example, `WHERE cust.country IN ('US', 'UK, 'France')`
- **`MEMBER OF` comparisons**—For example, `WHERE acct MEMBER OF cust.accounts`
- **Built-in string functions**—`CONCAT(string, string)`, `SUBSTRING(string, start, length)`, `LOCATE(string, string)`, and `LENGTH(string)`
- **Built-in arithmetic functions**—`ABS(number)`, `MOD(int, int)`, and `SQRT(double)`

The `WHERE` clause can also use input parameters, using the syntax `?1`, `?2`, and so forth. Parameters are numbered starting with *1* and are used in the order in which they appear in the find or select method signature. For example, consider the clause `WHERE acct.balance > ?1`. At runtime, the query processor substitutes the values of the parameters provided to the find or select method into the query.

The `ORDER BY` clause tells the query processor to order the results in a `Collection` by a particular CMP field or set of CMP fields. If the return type of the query is a bean type—that is, the `SELECT` clause has `OBJECT(cust)`—such as for find methods, the CMP fields used for ordering must be valid fields of the returned bean type. If the return type of the query is a CMP field type, the `ORDER BY` clause must use the same CMP field. Ordering can be in ascending or descending order. For example, consider the following clause:

```
ORDER BY cust.firstName ASC, cust.lastName DESC
```

This clause returns a `Collection` of customers in ascending order by their first names. Those customers within the `Collection` having the same first name are in descending order by their last names. Ascending order is the default if the `ORDER BY`

clause specifies neither `ASC` nor `DESC`. In addition, all null values in the returned `Collection` either precede or follow all non-null values.

We present more EJB QL queries when we look at the entity bean example application in Chapter 8.

Although EJB QL offers certain advantages, the language itself is not quite as rich as SQL. In particular, EJB QL cannot form some of the sophisticated queries of which SQL is capable, such as nested queries. However, it is likely that application server query tools will handle these differences between the languages. Future versions of the EJB specification will enhance the richness of EJB QL.

Bean-Managed Persistence

When it uses BMP, an entity bean uses a resource manager–specific interface (API) to access state. (In the BMP approach to managing entity object state persistence, the entity bean itself manages the access to the underlying state in a resource manager.)

Figure 7.4 shows three entity bean classes (`AccountBean`, `AccountBean2`, and `AccountBean3`). Each class accesses a different resource manager type. For example, a bean uses JDBC to access state stored in a relational database. In Figure 7.4, `AccountBean` uses the JDBC API to access state stored in the relational database management system (RDBMS). If the state is stored in a different type of database, the bean uses a different API to access the state, and the API is specific to the resource manager. Thus, `AccountBean2` uses an API specific to the RM2 adapter for its RM2-type database. This means that if an entity bean uses BMP, the bean code is, in general, dependent on the type of the resource manager.

For example, the `AccountBean` entity bean class may use JDBC to access the state of the Account entity objects in a relational database, as Code Example 7.8 illustrates.

In the example, the implementation of the `debit` method obtains the primary key of the Account entity object currently associated with the instance. The method uses the JDBC API to update the account balance. Note that in a real-life application development, the bean developer would likely use some data access tools, such as command beans, on top of JDBC rather than coding JDBC directly in the entity bean class.

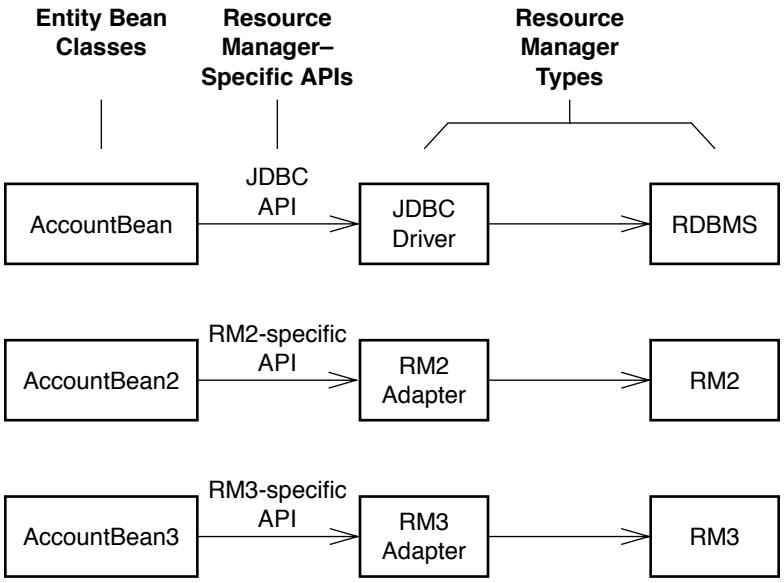

Figure 7.4 Entity Beans Using BMP to Access Different Resource Managers

```
public class AccountBean implements EntityBean {
      ...
      public void debit(double amount) {
         Connection con = ...;
         PreparedStatement pstmt = con.prepareStatement(
            "UPDATE Account SET acct_balance = acct_balance - ? " +
            "WHERE acct_number = ?"
         );
         pstmt.setDouble(1, amount);
         pstmt.setString(2, (String)ctx.getPrimaryKey());
         pstmt.execute();
         con.close();
      }
   }
```

Code Example 7.8 Using the JDBC API to Access State

BMP's main advantage is that it simplifies deploying the entity bean. When an entity bean uses BMP, no deployment tasks are necessary to adapt the bean to the resource manager type or to the database schema used within the resource manager. At the same time, this is also the main disadvantage of BMP, because an entity bean using BMP is, in general, dependent on the resource manager type and the database schema. This dependency makes the entity bean less reusable across different operational environments and also means that the developer has to write much more code to do the resource manager access.

However, an entity bean using BMP can achieve some degree of independence of the entity bean code from the resource manager type and the database schema. This can be accomplished, for example, by using portable data access components when developing the BMP entity bean. Essentially, the entity bean class uses the data access components to access the entity object's state. The data access components would provide deployment interfaces for customizing the data access logic to different database schemas or even to a different resource manager type, without requiring changes to the entity bean's code.

Figure 7.5 shows the AccountBean entity bean using three APIs to access three resource manager types. This example is very much like Figure 7.4, with one significant difference. Note that instead of three separate entity beans classes (`AccountBean`, `AccountBean2`, and `AccountBean3`) implementing access to the resource manager APIs, a single entity bean class, `AccountBean`, using data access components, can access the different resource manager–specific APIs. The data access components support all three resource manager types.

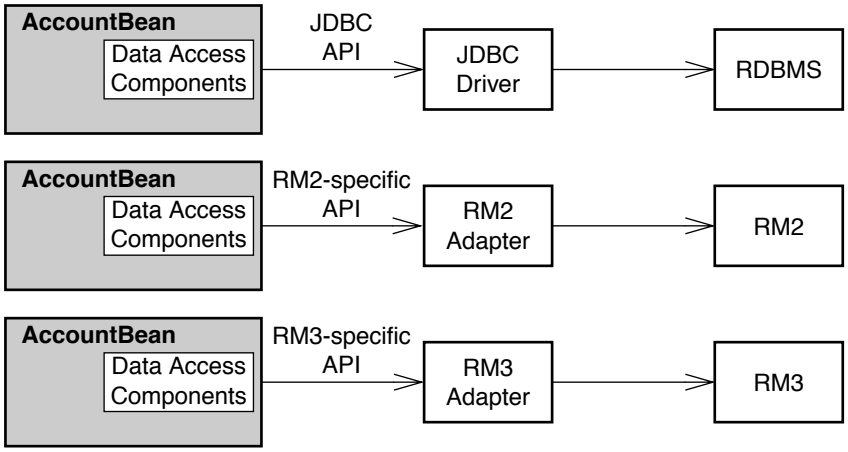

Figure 7.5 Entity Beans Using Data Access Components to Access Resource Managers

On the surface, this is similar to CMP. There is, however, a significant difference between CMP and the data access component approach. The CMP approach allows the EJB container to provide a sophisticated persistence manager that can cache the entity object's state in the container. The caching strategy implemented by the persistence manager can be tuned without making modifications to the entity bean's code. It is important to note that the CMP cache may be shared by multiple instances of the same entity bean or even by instances of different entity beans. This contrasts with the data access components approach, in which it is not possible, in general, to build a cache that can be shared by multiple instances.

7.2.2 Entity Bean Class Methods

The bean developer's primary focus is the development of the entity bean class. The bean developer is responsible for writing the implementation of the following methods:

- Business methods from the component interface
- Create methods from the home interface
- For BMP entity beans, find methods from the home interface
- Business methods from the home interface
- The container callback methods defined in the `javax.ejb.EntityBean` interface

In addition, for entity beans with CMP, the bean developer needs to write `abstract` get and set accessor methods for container-managed fields and relationships. The developer may also define `abstract ejbSelect` methods in the CMP bean implementation class for executing EJB QL queries internal to the bean.

The following subsections describe these methods in more detail. Section 7.2.3, Entity Bean Instance Life Cycle, on page 196 explains when and in what context the EJB container invokes these methods.

The implementation class for entity beans that use EJB 2.0 or 2.1 container-managed persistence must be an abstract class. The implementation class for entity beans that use bean-managed persistence is a concrete class.

Entity Bean Accessor Methods

Entity beans using CMP use accessor methods for the container-managed persistent fields. These accessor methods take the place of instance field declarations in the bean class. The get and set accessor methods are declared `public` and `abstract`. The bean class does not provide an implementation for these methods, because the EJB container does so. Accessor methods must be declared for both CMP and CMR fields.

We've already shown AccountBean's get and set methods for its container-managed fields `accountNumber` and `balance`. In addition, AccountBean might maintain a many-to-one relationship between accounts and customers. An account might also be involved in a one-to-many relationship, such as when a single account has a set of detail lines associated to it. If these relationships were implemented as container-managed relationships, AccountBean would include accessor methods for these CMR fields, too (Code Example 7.9):

```
public abstract class AccountBean implements EntityBean {
   // Accessor methods for container-managed fields

   public abstract String getAccountNumber();    // accountNumber
   public abstract void setAccountNumber(String v);

   public abstract double getBalance();         // balance
   public abstract void setBalance(double v);
   ...
   // Accessor methods for container-managed relationship fields
   public abstract Customer getCustomer();      // Customer
   public abstract void setCustomer(Customer customer);

   public abstract Collection getAcctDetails(); // Detail lines
   public abstract void setAcctDetails(Collection acctDetails);
}
```

Code Example 7.9 AccountBean Entity Bean

Note that the `getCustomer` accessor method returns the local entity object type `Customer`, whereas `getAcctDetails` returns a `Collection` of `Detail` objects.

Entity Bean Business Methods

The bean developer implements in the entity bean class the business methods declared by the entity bean's component interface. The rules for the business method implementations of an entity bean are similar to those for a session bean implementation. The number and types of parameters and the return value type for these business methods must match those defined in the component interface. In addition, the `throws` clause for the entity bean class business methods must not include more checked exceptions than the `throws` clause of the corresponding component interface methods. (Note that the methods in the entity bean class can define fewer exceptions than the methods in the remote interface.) Note too that the business methods must be declared `public`. They must not be declared `final` or `static`.

Recall that the `Account` component interface defines the following business methods:

```
double getBalance();
void credit(double amount);
void debit(double amount) throws InsufficientFundsException;
```

The bean developer implements these same business methods in the `AccountBean` class, as follows:

```
public double getBalance() { ... }
public void credit(double amount) { ... }
public void debit(double amount)
    throws InsufficientFundsException { ... }
```

Entity Bean Create Methods

The entity bean class defines `ejbCreate` and `ejbPostCreate` methods that correspond to the create methods defined in the home interface. For each `create` method in the home interface, the bean developer implements an `ejbCreate` and an `ejbPostCreate` method in the entity bean class. Remember that an entity bean may choose *not* to expose the create functionality to clients, in which case the entity bean home interface defines no create methods, and, of course, the bean developer does not implement `ejbCreate` or `ejbPostCreate` methods in the bean class.

The `ejbCreate` and `ejbPostCreate` methods have the same number of parameters, each of which must be of the same type as those defined in the home interface's corresponding create method. However, the `ejbCreate` methods differ from

the create methods, defining the bean's primary key type as their return value type. The `ejbPostCreate` defines the return type as `void`. The `throws` clause for each `ejbCreate` and `ejbPostCreate` method must not include more checked exceptions than the `throws` clause of the corresponding create method. However, the `ejbCreate` method `throws` clause can have fewer exceptions than the corresponding create method.

Like the business methods, the `ejbCreate` methods must be declared `public`. They must not be declared `final` or `static`. For example, the `Account` home interface declares the following create methods:

```
Account create(String lastName, String firstName)
    throws CreateException, BadNameException;
Account create(String lastName)
    throws CreateException;
Account createBusinessAcct(String businessName)
    throws CreateException;
```

Note that if `Account`'s home interface had implemented a remote view, each of its create methods would have thrown a `RemoteException` in addition to the other exceptions.

The bean developer implements these corresponding `ejbCreate` and `ejbPostCreate` methods in the `AccountBean` class:

```
public AccountKey ejbCreate(String lastName, String firstName)
    throws BadNameException { .... }
public void ejbPostCreate(String lastName, String firstName)
    throws BadNameException { .... }

public AccountKey ejbCreate(String lastName) { .... }
public void ejbPostCreate(String lastName) { .... }

public AccountKey ejbCreate(String businessName) { .... }
public void ejbPostCreate(String businessName) { .... }
```

When creating a new entity instance, the EJB container first invokes the `ejbCreate` method and then invokes the matching `ejbPostCreate` method (see the section Invocation of Create Methods on page 199).

The `ejbPostCreate` method can be used for any initialization work that requires the identity of the newly created entity bean in the form of its `EJBObject` or `EJBLocalObject` reference. If it needs to pass a reference of the entity object that is being created to another enterprise bean, an instance must do so in the `ejb-`

PostCreate method, not in the ejbCreate method. For example, the ejbPostCreate method may pass the created Account object to the Customer object as an argument in the addAccount method:

```
public class AccountBean implements EntityBean {
    EntityContext ctx;

    public ejbCreate(Customer cust) {
        // This would be an ERROR because it is illegal to
        // invoke ctx.getEJBLocalObject from an ejbCreate method.
        cust.addAccount((Account)ctx.getEJBLocalObject());
        ...
    }
    public ejbPostCreate(Customer cust) {
        // This is correct.
        cust.addAccount((Account)ctx.getEJBLocalObject());
    }
    ...
}
```

Entity Bean Find Methods

For entity beans with bean-managed persistence, the bean developer must also implement in the entity bean class ejbFind methods that correspond to the find methods defined in the home interface. Recall that an entity bean's home interface defines one or more find methods, which a client uses to locate entity objects. For CMP entity beans, the ejbFind methods are generated by the container using the EJB QL query provided by the bean developer, so the bean developer does not need to write the implementations of ejbFind methods for CMP entity beans.

At a minimum, the developer of a BMP entity bean implements an ejbFindByPrimaryKey method corresponding to the findByPrimaryKey method, which is defined by all entity bean home interfaces. This method looks up a single entity object, using the object's primary key. The developer also implements ejbFind methods that correspond to any additional find methods, such as those that return multiple objects, defined by the home interface. Each find method implementation has the same number of parameters, and each parameter is of the same type as the corresponding home interface find method.

The entity bean class implementations of the find methods differ in some respects from the home interface definition. In the home interface, the result type

of find methods that return single objects, whether the `findByPrimaryKey` or another find method, is the entity bean's component interface. The result type of the corresponding `ejbFind` methods is the entity bean's primary key type. Similarly, in the home interface, the result type of find methods returning multiple objects is a collection (`java.util.Collection`) of objects implementing the component interface. In the implementation class, the result type of the corresponding `ejbFind` methods that return multiple objects is a collection (`java.util.Collection`) of objects of the bean's primary key type.

Finally, similar to the create methods, the `throws` clause for each `ejbFind` method must not include more checked exceptions than the `throws` clause of the corresponding find method. However, the `ejbFind` method `throws` clause can have fewer exceptions than the corresponding find method. The `ejbFind` methods must be declared `public`. They must not be declared `final` or `static`.

For example, the `AccountHome` interface defines the `findByPrimaryKey` method and an additional method, `findInactive`, which returns multiple objects:

```
import java.util.Collection;
Account findByPrimaryKey(AccountKey primaryKey)
    throws FinderException;
Collection findInactive(Date sinceWhen)
    throws FinderException, BadDateException;
```

If the AccountEJB bean is a BMP entity bean, the bean developer implements these methods in the entity bean class as follows:

```
public AccountPrimaryKey ejbFindByPrimaryKey
    (AccountPrimaryKey primkey) throws FinderException { ... };
public Collection ejbFindInactive(Date sinceWhen)
    throws BadDateException { ... };
```

Entity Bean Home Methods

Entity bean classes must also provide an implementation for each home method listed in the home interface. The bean developer implements these home methods with corresponding ejbHome methods. For example, the `AccountHome` interface includes one home method:

```
public void debitAcctFee (float fee_amt) throws OutOfRangeException;
```

The bean developer implements this method in the entity bean class as follows:

```
public void ejbHomeDebitAcctFee (float fee_amt)
    throws OutOfRangeException { ... };
```

Home methods are similar to static methods in Java classes in that they do not operate on a specific entity bean instance that has an identity. Home methods are executed by a bean instance that has not been assigned an identity. As a result, home methods cannot access the identity of the bean by using the `getPrimaryKey`, `getEJBObject`, and `getEJBLocalObject` methods of the `EntityContext` interface. This also means that home methods cannot call the get or set accessor methods for CMP and CMR fields. However, home methods can call the `ejbSelect` methods defined in the bean class. This allows home methods to execute queries and to process the results of the queries.

Select Methods

Select methods, available only for CMP entity beans, are similar to find methods in that the bean developer provides the method definitions, but the methods themselves are implemented using EJB QL queries. The bean developer defines select methods as abstract methods in the bean implementation class with a name of the form `ejbSelect<METHOD>`. The developer also provides an EJB QL query for the `ejbSelect` method in the bean's deployment descriptor. At deployment, the container's tools generate the implementation code for the `ejbSelect` method, using the EJB QL query provided by the developer.

Keep in mind that `ejbSelect` methods are not exposed through the home or component interfaces. Hence the client cannot invoke `ejbSelect` methods directly. The developer may define a home method on the home interface that delegates to the `ejbSelect` method, and then the client can invoke that home method.

However, unlike for a find method, the return value of an `ejbSelect` method is not restricted to the bean's component interface. In fact, `ejbSelect` methods can return any type, including the types of the CMP fields in the bean and the local objects of related beans. The return value may also be a single value or a `Collection` of values.

For example, an example of an `ejbSelect` method definition for the AccountBean follows:

```
public abstract Collection ejbSelectInactiveCustomers
       (Date sinceWhen);
```

This `ejbSelectInactiveCustomers` method returns a `java.util.Collection` object containing the local entity objects for customers whose accounts have been inactive since the date provided in the parameter to the method.

EntityBean Interface Methods

An entity bean class is required to implement the methods defined by the `javax.ejb.EntityBean` interface. The EJB container invokes these methods on the bean instance at specific points in an entity bean instance's life cycle. Code Example 7.10 shows the definition of the `EntityBean` interface methods:

```
public interface EntityBean extends EnterpriseBean {
   public void setEntityContext(EntityContext ctx)
      throws EJBException, RemoteException;
   public void unsetEntityContext()
      throws EJBException, RemoteException;
   public void ejbRemove()
      throws RemoveException, EJBException, RemoteException;
   public void ejbActivate() throws EJBException, RemoteException;
   public void ejbPassivate() throws EJBException, RemoteException;
   public void ejbLoad() throws EJBException, RemoteException;
   public void ejbStore() throws EJBException, RemoteException;
}
```

Code Example 7.10 `EntityBean` Interface

Note that although the `EntityBean` interface methods throw `RemoteException`, the EJB 2.1 specification mandates that implementations of these methods must not throw `RemoteException`.

7.2.3 Entity Bean Instance Life Cycle

Every entity bean instance has a life cycle that starts from its time of creation and continues through its removal. The EJB container manages the life cycle for every instance of an entity bean class.

Bean developers who manually code the entity bean class need to know what happens during an entity bean's life cycle and how that life cycle is managed by the container. On the other hand, bean developers who use an EJB-aware application development tool do not need to know most of this information, because the application development tool may provide a simpler abstraction.

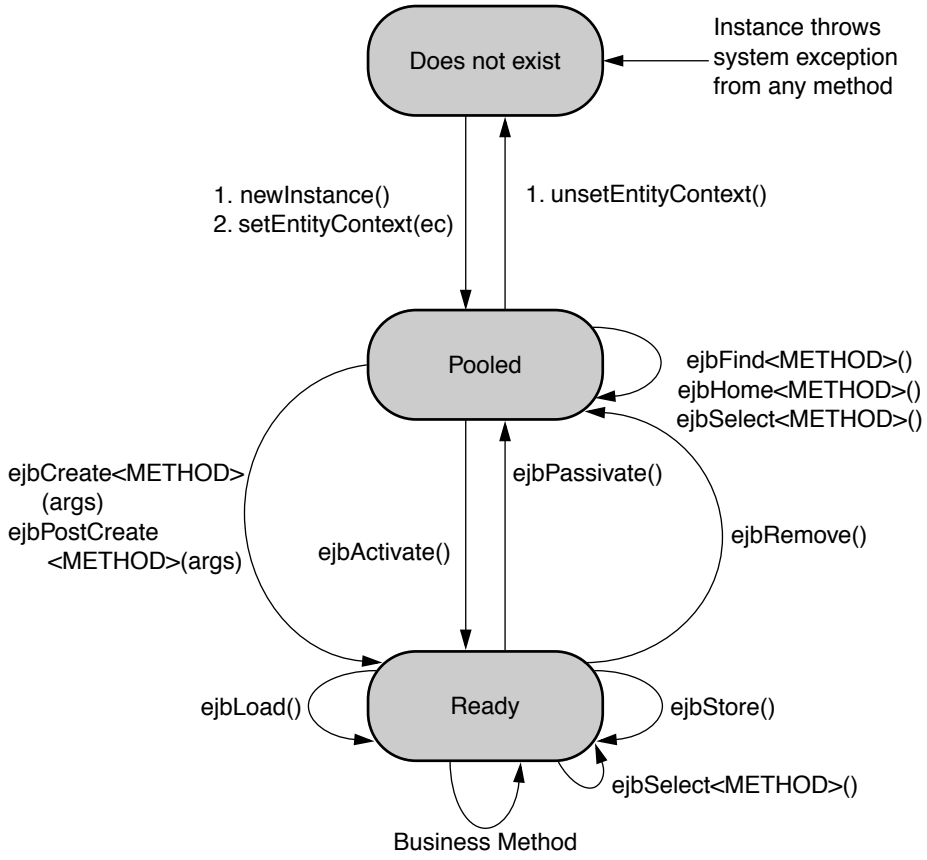

Figure 7.6 Life Cycle of an Entity Bean Instance

This section describes the various stages in the life cycle of entity bean instances, as well as the persistence and transaction management interaction details between the container and entity bean instances. These details are useful for advanced developers trying to tune the performance of their entity beans.

Figure 7.6 illustrates the life cycle of an entity bean instance. The diagram shows that an entity bean instance is in one of the following three states:

1. **Does not exist**—The container has not yet created an instance, or the container has discarded the instance.

2. **Pooled**—An instance is in the pooled state when it is not associated with a particular entity object identity.
3. **Ready**—An instance is in the ready state when the instance is associated with an entity object identity.

Each entity instance state has defined characteristics. In addition, transitions between states happen as a result of certain actions. The EJB container drives the state transition in response to client-invoked methods from the home and remote interfaces and in response to container-internal events, such as transaction commit, exceptions, or resource management.

- An instance's life begins when the container creates the instance by using `newInstance` or the `new` operator. The container then invokes the `setEntityContext` method to pass the instance a reference to its `EntityContext` object. At this point, the instance transitions to the pooled state.
- While in the pooled state, the instance is not associated with an entity object identity. As a result, all instances in the pooled state are considered equivalent. The container can perform the following actions with an instance in the pooled state:
  - Execute an `ejbFind<METHOD>`, `ejbSelect<METHOD>`, or `ejbHome<METHOD>` method in response to a client-invoked find or home method through the home interface. Note that the instance does not move to the ready state during the execution of the `ejbFind<METHOD>`, `ejbSelect<METHOD>`, or `ejbHome<METHOD>` method.
  - Associate the instance with an existing entity object by invoking the `ejbActivate` method on the instance. A successful execution of the `ejbActivate` method transitions the instance to the ready state.
  - Use the instance to create a new entity object by invoking the `ejbCreate` and `ejbPostCreate` methods on the instance. Successfully executing the `ejbCreate` and `ejbPostCreate` methods transitions the instance to the ready state.
  - Discard the instance by invoking the `unsetEntityContext` method on the instance. By doing so, the container notifies the instance that it will invoke no other methods on the instance.
- When the instance is in the ready state, it has an identity—that is, a primary key—and is associated with an entity object. The container can perform the

following actions on an instance in the ready state:

- Invoke the `ejbLoad` method on the instance. The `ejbLoad` method instructs the instance to synchronize any cached representation it maintains of the entity object's state from the entity object's state in the resource manager. The instance remains in the ready state.
- Invoke the `ejbStore` method on the instance. The `ejbStore` method instructs the instance to synchronize the entity object's state in the resource manager with updates to the state that may have been cached in the instance. The instance remains in the ready state.
- Invoke a business method in response to a client-invoked method in the component interface. The instance remains in the ready state.
- Invoke the `ejbPassivate` method on the instance and move it to the pooled state.
- Invoke the `ejbRemove` method on the instance in response to a client-invoked `remove` method. The `ejbRemove` method transitions the instance to the pooled state.

- If an instance throws and does not catch a system exception—that is, an exception that is a subclass of `java.lang.RuntimeException`—the container catches the exception and transitions the instance to the does-not-exist state, regardless of the bean's original state.

Invocation of Create Methods

This section uses object interaction diagrams (OIDs) to illustrate what happens when an entity object is created with BMP or CMP. Figure 7.7 illustrates creating an entity object with BMP.

1. The client starts a transaction, using the `begin` method of the `UserTransaction` interface.
2. The client invokes a `create` method on the home object, which is implemented by the container. Keep in mind that the EJB container performs the diagramed operations attributed to the home object.
3. The home object invokes the matching `ejbCreate` method on the entity bean instance.
4. The bean instance creates a representation of the entity object state in the database.

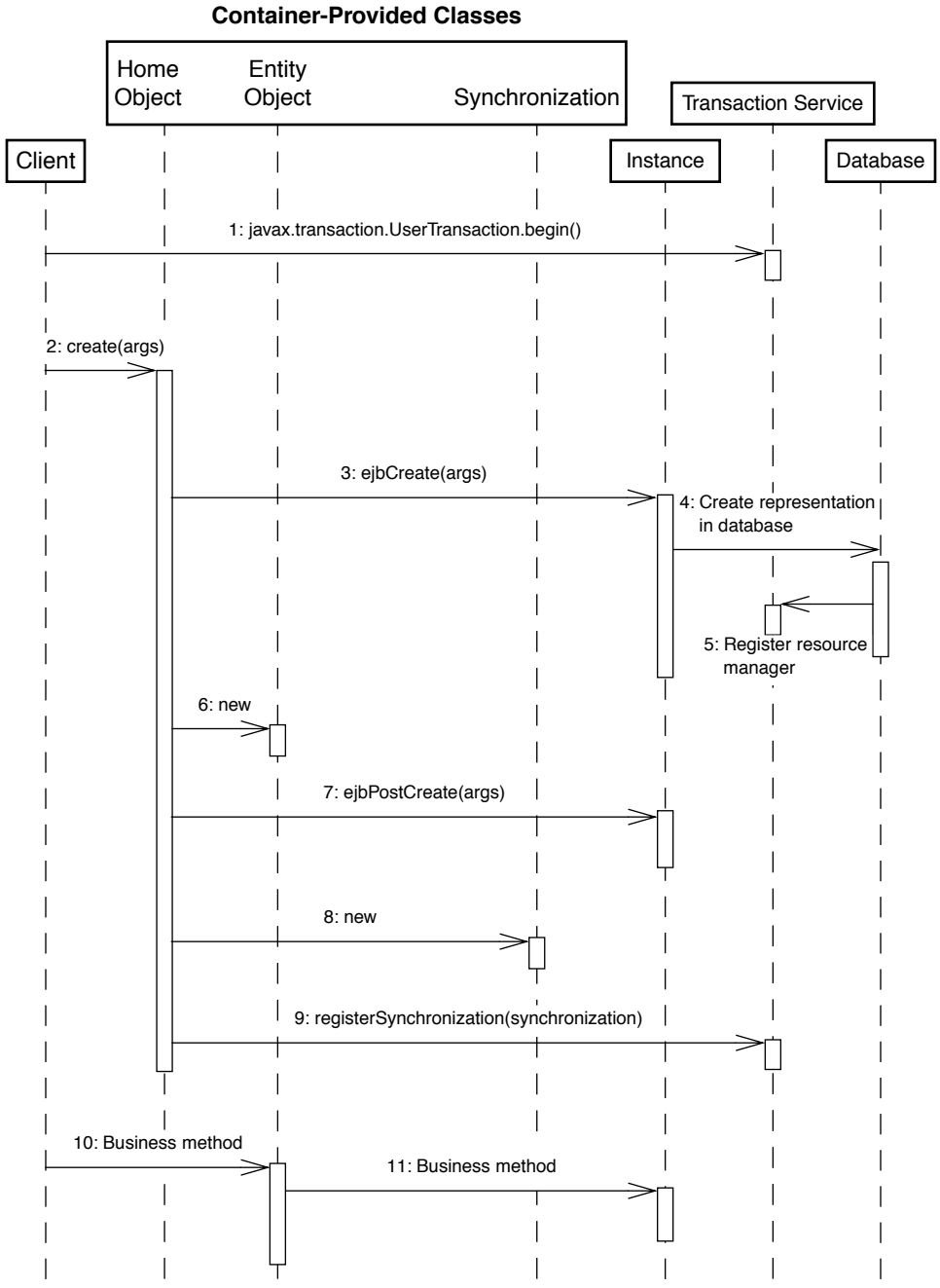

Figure 7.7 OID of Creation of Entity Object with BMP

5. The database system registers itself with the transaction service, which is a synonym for *transaction manager*.
6. The home object creates the entity object, which may be a distributed object that implements the entity bean's remote interface or a local object that implements the entity bean's local interface.
7. The home object invokes the matching `ejbPostCreate` method on the bean instance.
8. The home object creates a transaction synchronization object.
9. The home object registers the transaction synchronization object with the transaction service.
10. The client invokes a business method on the newly created entity object in the same transaction context as the `create` method.
11. The entity object delegates the invocation to the bean instance.

When the client eventually attempts to commit the transaction, the transaction service orchestrates the commit protocol, as described in the section Transaction-Commit OID on page 210.

Note that creation of the entity object is considered part of the transaction. If the transaction fails, the representation of the object's state in the database is automatically deleted, and the entity object does not exist.

Figure 7.8 shows the OID for the creation of an entity object with CMP.

1. The client starts a transaction by using the `begin` method of the `UserTransaction` interface.
2. The client invokes a `create` method on the home object. As noted earlier, the EJB container performs the diagramed operations attributed to the home object.
3. The home object invokes the matching `ejbCreate` method on the entity bean instance. The bean instance initializes itself, using the data passed in the arguments of the `ejbCreate` method.
4. The home object uses the CMP state to create the representation of the entity object state in the database.
5. The database system registers itself with the transaction service.

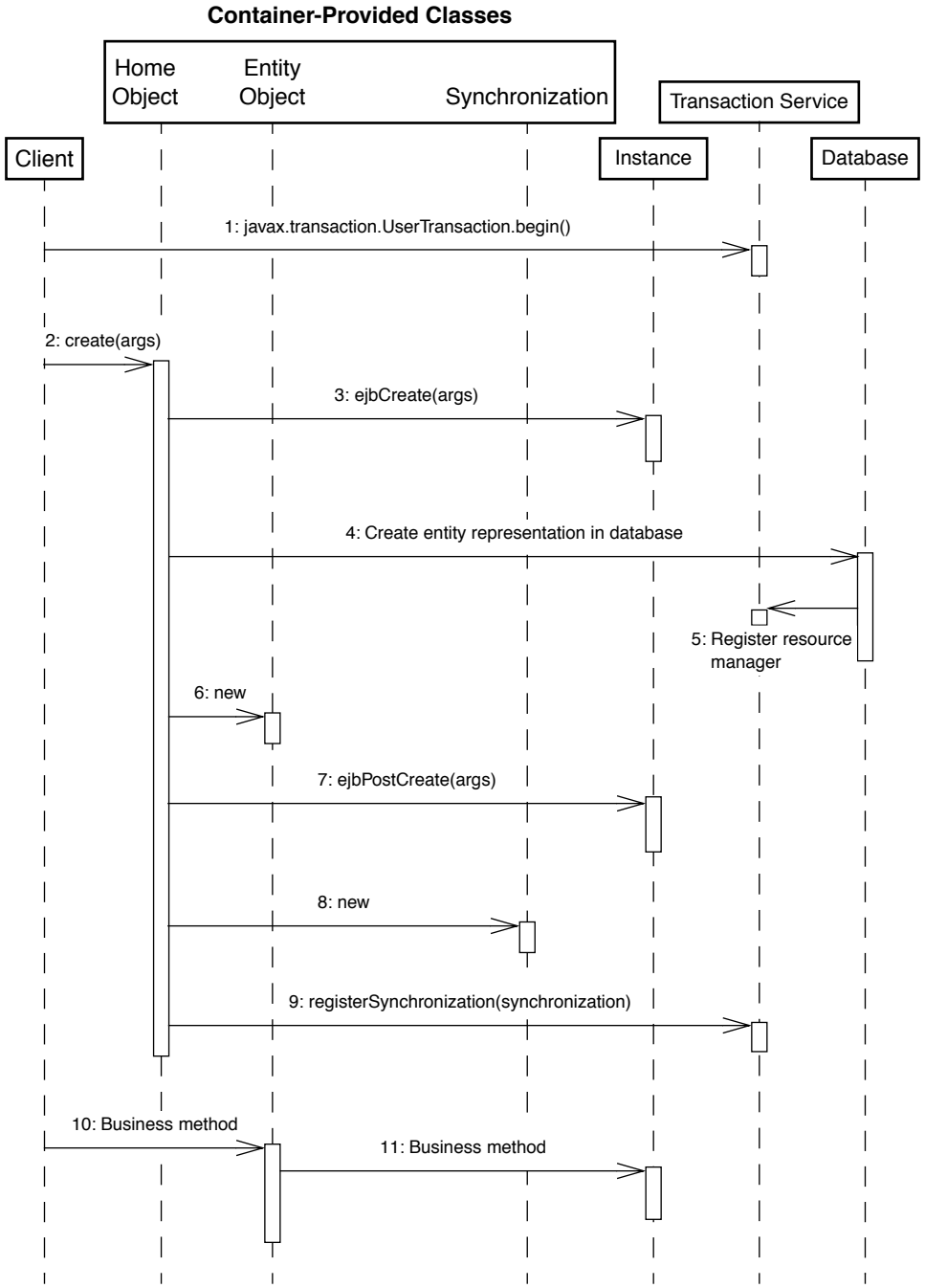

Figure 7.8 OID of Creation of Entity Object with CMP

6. The home object creates the entity object. The entity object implements the entity bean's component interface.
7. The home object invokes the matching `ejbPostCreate` method on the bean instance.
8. The home object creates a transaction synchronization object.
9. The home object registers the transaction synchronization object with the transaction service.
10. The client invokes a business method on the newly created entity object in the same transaction context as the `create` method.
11. The entity object delegates the invocation to the bean instance.

Invocation of the `remove` Method

The next two figures illustrate removing entity objects. Figure 7.9 shows the steps that take place when removing an entity object with BMP, and Figure 7.10 shows the equivalent steps for removing an entity object with CMP.

1. In Figure 7.9, the client invokes the `remove` method on the entity object.
2. The entity object invokes the `ejbRemove` method on the bean instance.
3. The instance removes the representation of the entity object state from the database.

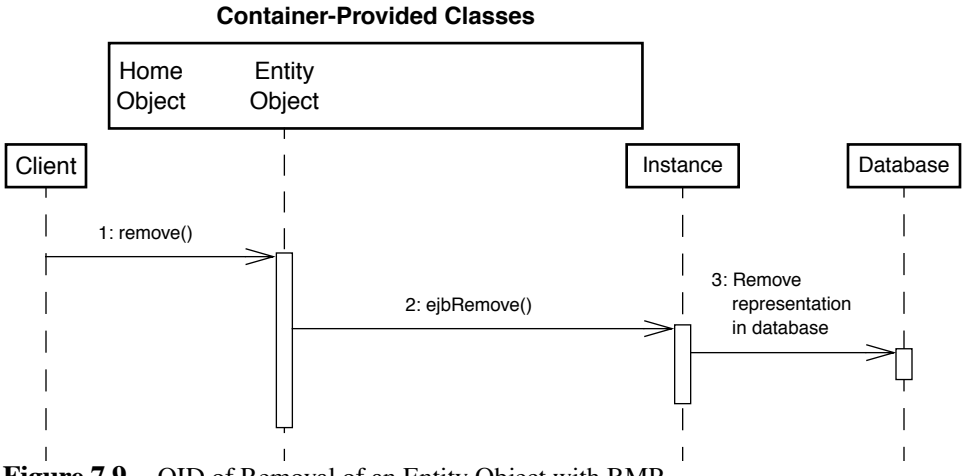

Figure 7.9 OID of Removal of an Entity Object with BMP

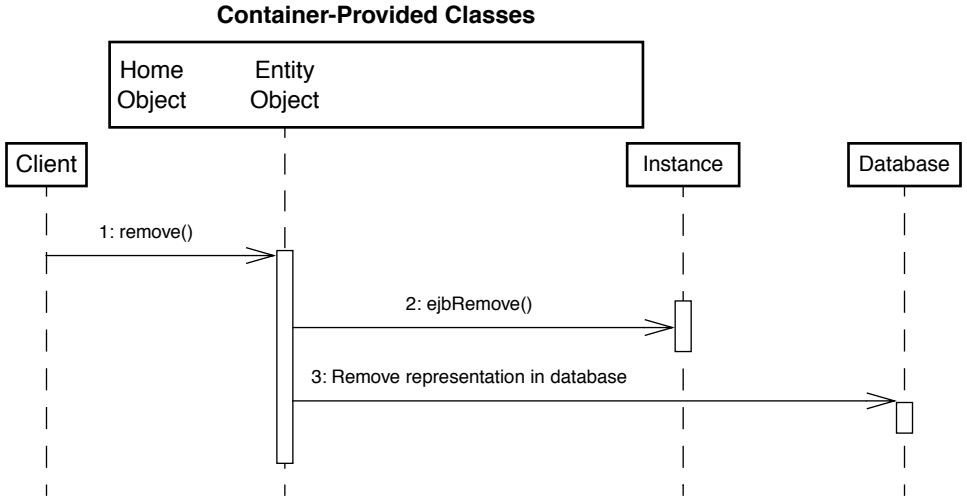

Figure 7.10 OID of Removal of Entity Object with CMP

Note that the diagram does not illustrate the transaction-related interactions among the container, transaction service, and database. The removal of the object's state representation from the database—step 3 in the diagram—is included as part of the transaction in which the `remove` method is executed. If the transaction fails, the object's state is not removed from the database, and the entity object continues to exist.

1. In Figure 7.10, the client invokes the `remove` method on the entity object.
2. The entity object invokes the `ejbRemove` method on the bean instance.
3. The entity object removes the representation of its state from the database.

Invocation of Find Methods

This section illustrates find method invocations on entity beans. Figure 7.11 shows the OID for a find method invocation on an entity bean instance with BMP. In contrast, Figure 7.12 shows the execution of a find method on an entity instance with CMP.

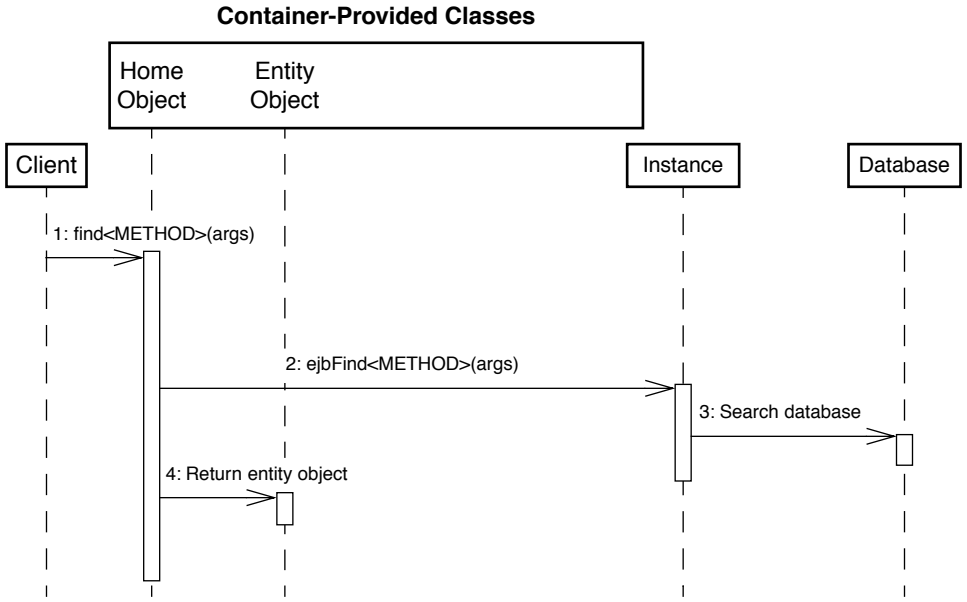

Figure 7.11 OID of Find Method Execution on a Bean-Managed Entity Instance

1. The client invokes a `find` method on the home object (Figure 7.11). As noted, the EJB container performs the diagramed operations attributed to the home object.
2. The home object invokes the matching `ejbFind` method on a bean instance.
3. The bean instance searches the database to find the object(s) that matches the find method's criteria. The instance returns the primary key—or collection of keys—from the `ejbFind` method.
4. The home object converts the returned primary key to an entity object reference—or collection of entity object references—and returns it to the client.

Note that the diagram does not illustrate the transaction-related interactions among the container, transaction service, and database. The database search—step 3 in the diagram—is included as part of the transaction in which the find method executes. Depending on the isolation level, the found objects may be protected from deletion by other transactions.

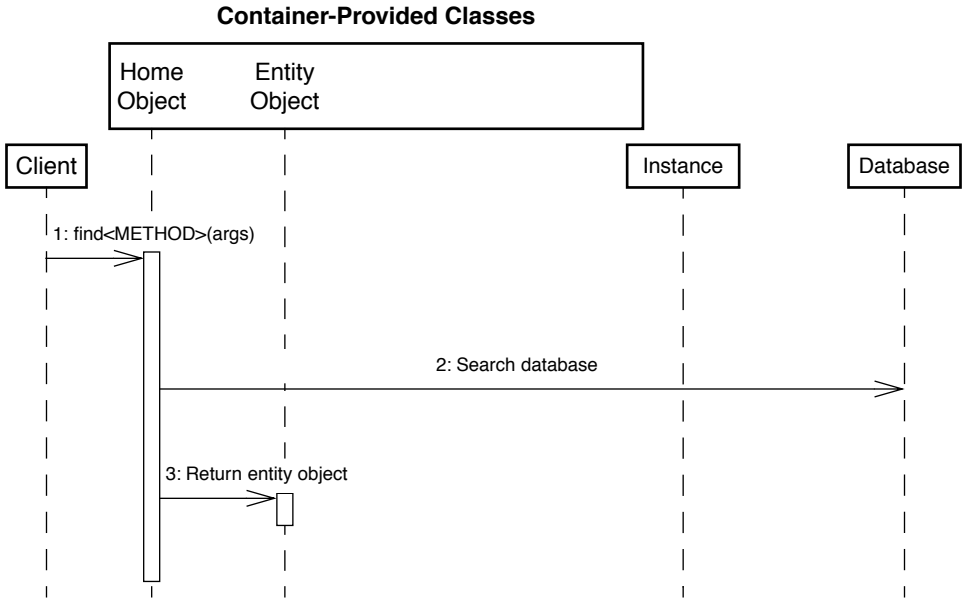

Figure 7.12 OID of Find Method Execution on a Container-Managed Entity Instance

1. The client invokes a `find` method on the home object (Figure 7.12). As noted, the EJB container performs the diagramed operations attributed to the home object.
2. The home object searches the database to find the object(s) that matches the find method's criteria.
3. The home object converts the primary key of the found object to an entity object reference and returns it to the client. If the find method finds more than one object, a collection of entity object references is returned to the client.

Passivation and Activation OID

The EJB architecture allows the EJB container to passivate an entity instance during a transaction. Figures 7.13 and 7.14 show the sequence of object interactions that occur for entity bean passivation and activation. Figure 7.13 is the OID for passivation and reactivation of an entity instance with BMP.

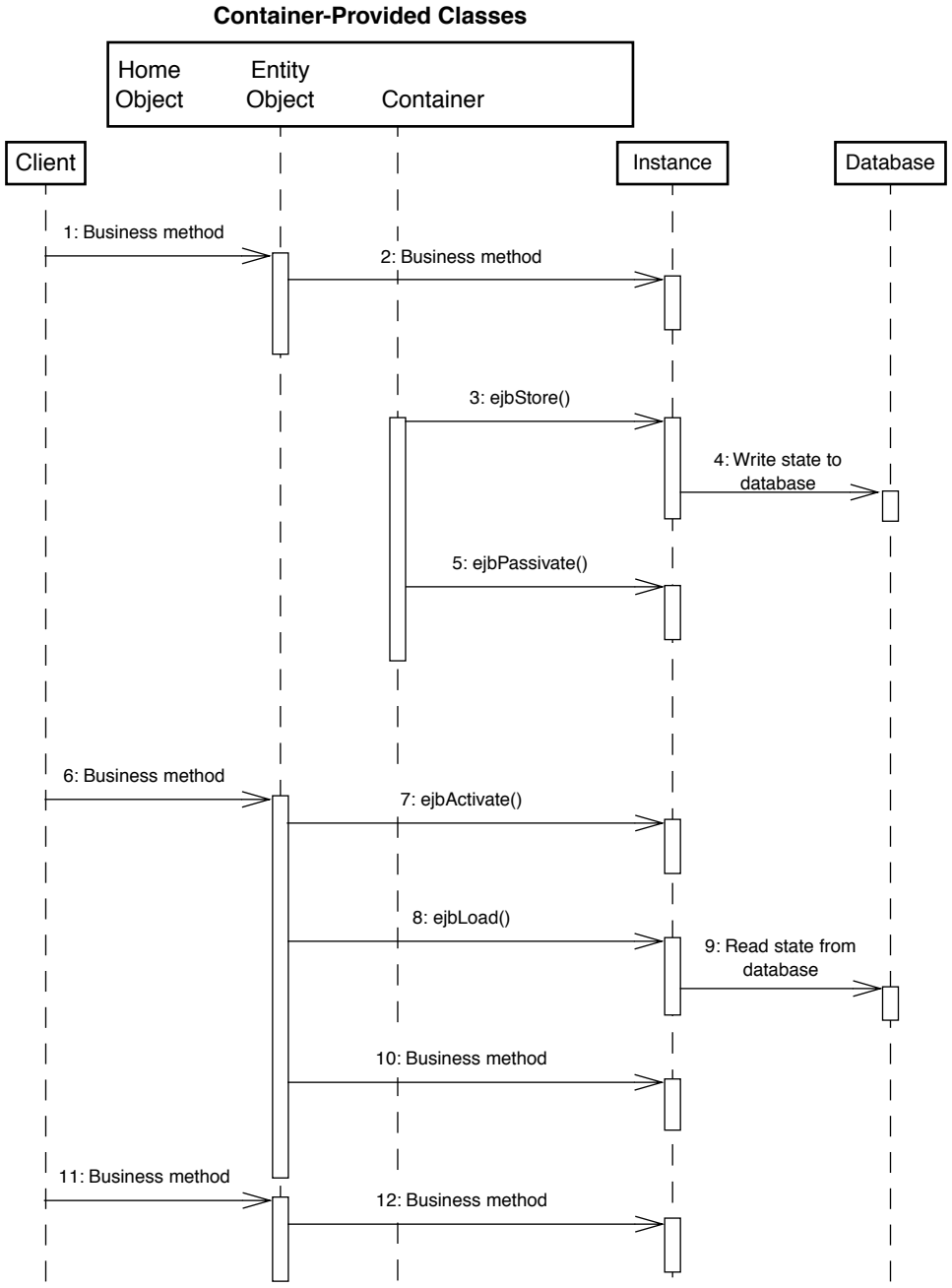

Figure 7.13 OID of Passivation and Reactivation of a Bean-Managed Entity Instance

1. The client invokes the last business method before passivation occurs.
2. The entity object delegates the business method to the bean instance.
3. The container decides to passivate the instance while it is associated with a transaction. (The reason for passivation may be, for example, that the container needs to reclaim the resources held by the instance.) The container invokes the `ejbStore` method on the instance.
4. The instance writes any cached updates made to the entity object state to the database.
5. The container invokes the `ejbPassivate` method in the instance. The instance should release into the pooled state any resources that it does not need. The instance is now in the pooled state. The container may use the instance to run find methods, activate the instance for another object identity, or release the instance through the `unsetEntityContext` method.
6. The client invokes another business method on the entity object.
7. The entity object allocates an instance from the pool. (The allocated instance could be the same instance or an instance different from the one that was associated with the object identity prior to passivation.) The container invokes the `ejbActivate` method in the instance.
8. The container invokes the `ejbLoad` method in the instance. Note that the container implements the entity object. Therefore, the `ejbActivate` and `ejbLoad` calls invoked in steps 7 and 8 are in fact invoked by the container.
9. The instance uses the `ejbLoad` method to read the object state, or parts of the object state, from the database.
10. The entity object delegates the business method to the instance.
11. The client invokes the next business method on the entity object.
12. Because the instance is in the ready state, the entity object can delegate the business method to the instance.

Figure 7.14 represents the same operations for an entity instance with CMP.

1. The client invokes the last business method before passivation occurs.
2. The entity object delegates the business method to the bean instance.

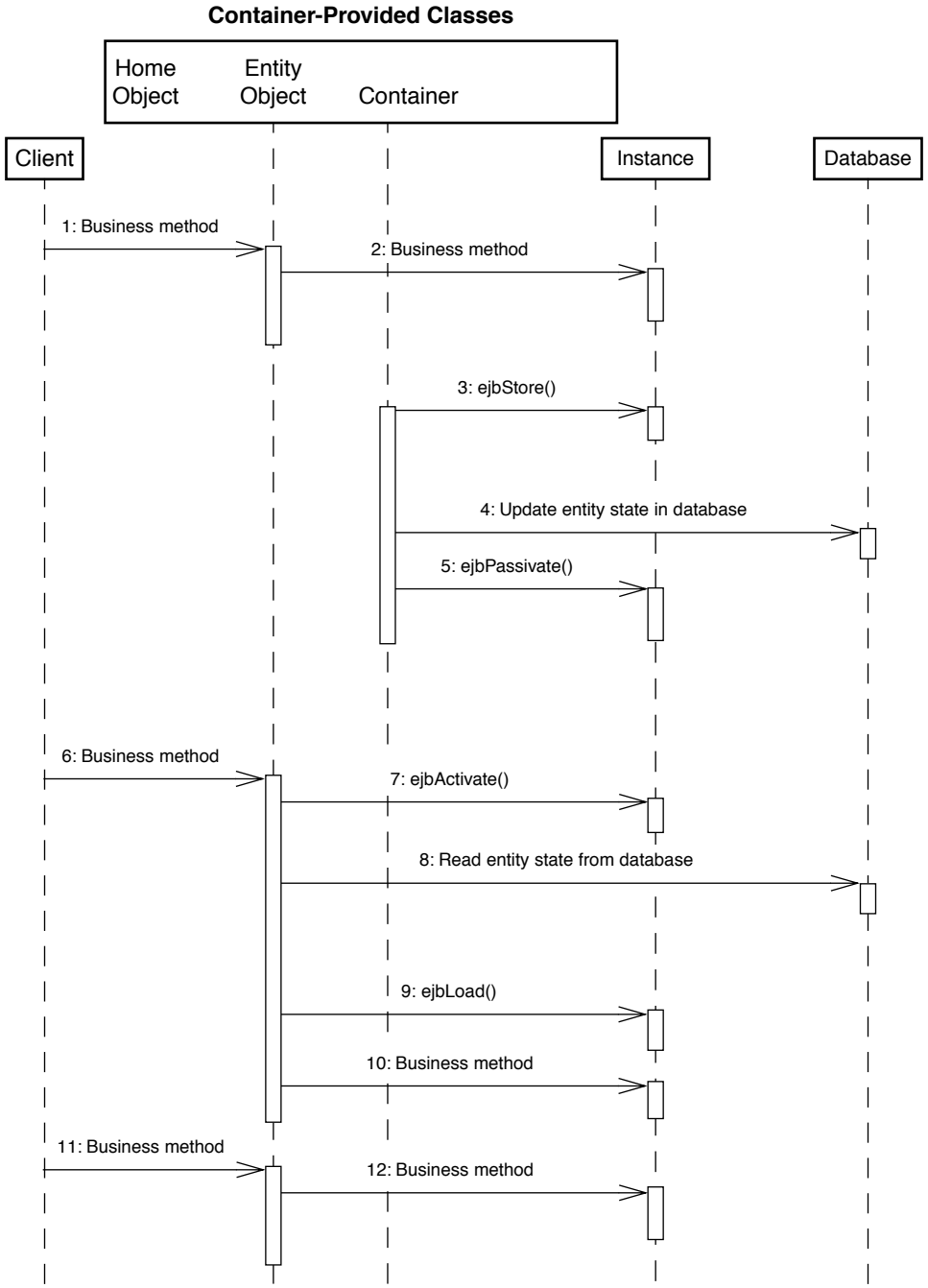

Figure 7.14 OID of Passivation and Reactivation of an Entity Instance with CMP

3. The container decides to passivate the instance while it is associated with a transaction. (The reason for passivation may be, for example, that the container needs to reclaim the resources held by the instance.) The container invokes the `ejbStore` method on the instance. The instance can use the `ejbStore` method to update its state. (See Section 7.2.4, Caching Entity Bean State, on page 219.)
4. The container updates the object state in the database with the extracted values of the CMP and CMR fields.
5. The container invokes the `ejbPassivate` method in the instance. The instance should release into the pooled state any resources that it does not need. The instance is now in the pooled state. The container may use the instance to run find methods, activate the instance for another object identity, or release the instance through the `unsetEntityContext` method.
6. The client invokes another business method on the entity object.
7. The entity object allocates an instance from the pool. (The instance could be the same instance or an instance different from the one that was associated with the object identity prior to passivation.) The container invokes the `ejbActivate` method in the instance.
8. The container reads the object state from the database.
9. The container invokes the `ejbLoad` method in the instance. Note that the container implements the entity object. Therefore, the `ejbActivate` and `ejbLoad` calls invoked in steps 7 and 9 (and the work in step 8) are in fact invoked by the container.
10. The entity object delegates the business method to the instance.
11. The client invokes the next business method on the entity object.
12. Because the instance is in the ready state, the entity object can delegate the business method to the instance.

Transaction-Commit OID

This section describes the operations that occur during transaction commit. Figure 7.15 shows the object interactions of the transaction-commit protocol for an entity instance with BMP.

1. After invoking methods on an entity bean, the client attempts to commit the transaction by invoking the `commit` method on the `UserTransaction` interface.

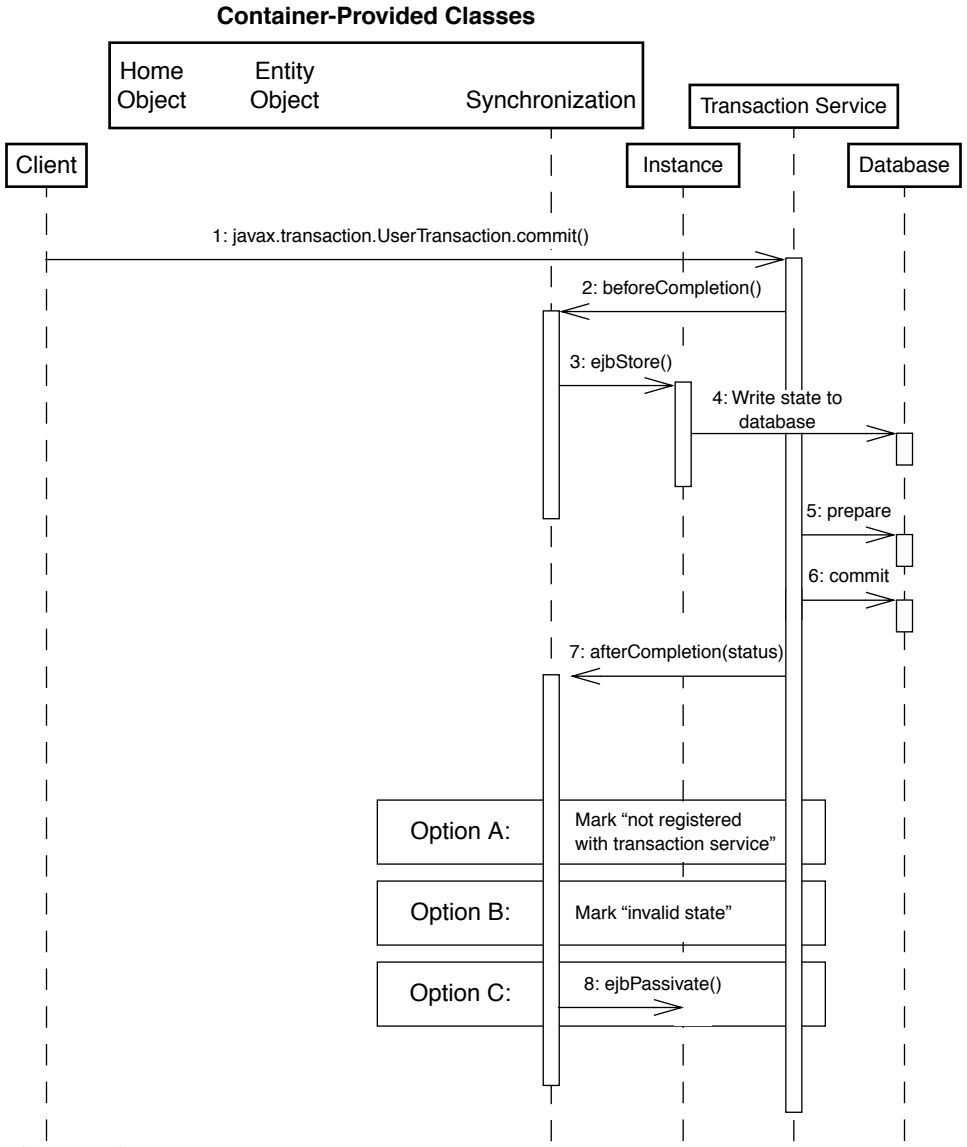

Figure 7.15 OID of Transaction Commit Protocol for a Bean-Managed Entity Instance

2. The transaction service invokes the `beforeCompletion` method on a transaction synchronization object implemented by the container.

3. The container invokes the `ejbStore` method on the bean instance used by the transaction.
4. The instance writes any cached updates made to the entity object state to the database.
5. The transaction service performs the prepare phase of the two-phase commit protocol. This step is skipped if the database is the only resource manager enlisted with the transaction and the transaction service implements the one-phase commit optimization.
6. The transaction service performs the commit phase of the two-phase commit protocol.
7. The transaction service invokes the `afterCompletion` method on the synchronization object.
8. If the container chooses commit option C—see the section Commit Options on page 217—the container invokes the `ejbPassivate` method in the instance.

Figure 7.16 shows the equivalent interactions of the transaction-commit protocol for an entity instance with CMP.

1. After invoking methods on an entity bean, the client attempts to commit the transaction by invoking the `commit` method on the `UserTransaction` interface.
2. The transaction service invokes the `beforeCompletion` method on a transaction synchronization object implemented by the container.
3. The container invokes the `ejbStore` method on the bean instance used by the transaction.
4. The container updates the object state in the database with the extracted values of the CMP and CMR fields.
5. The transaction service performs the prepare phase of the two-phase commit protocol. This step is skipped if the database is the only resource manager enlisted with the transaction and the transaction service implements the one-phase commit optimization.
6. The transaction service performs the commit phase of the two-phase commit protocol.
7. The transaction service invokes the `afterCompletion` method on the synchronization object.
8. If the container chooses commit option C, the container invokes the `ejbPassivate` method in the instance.

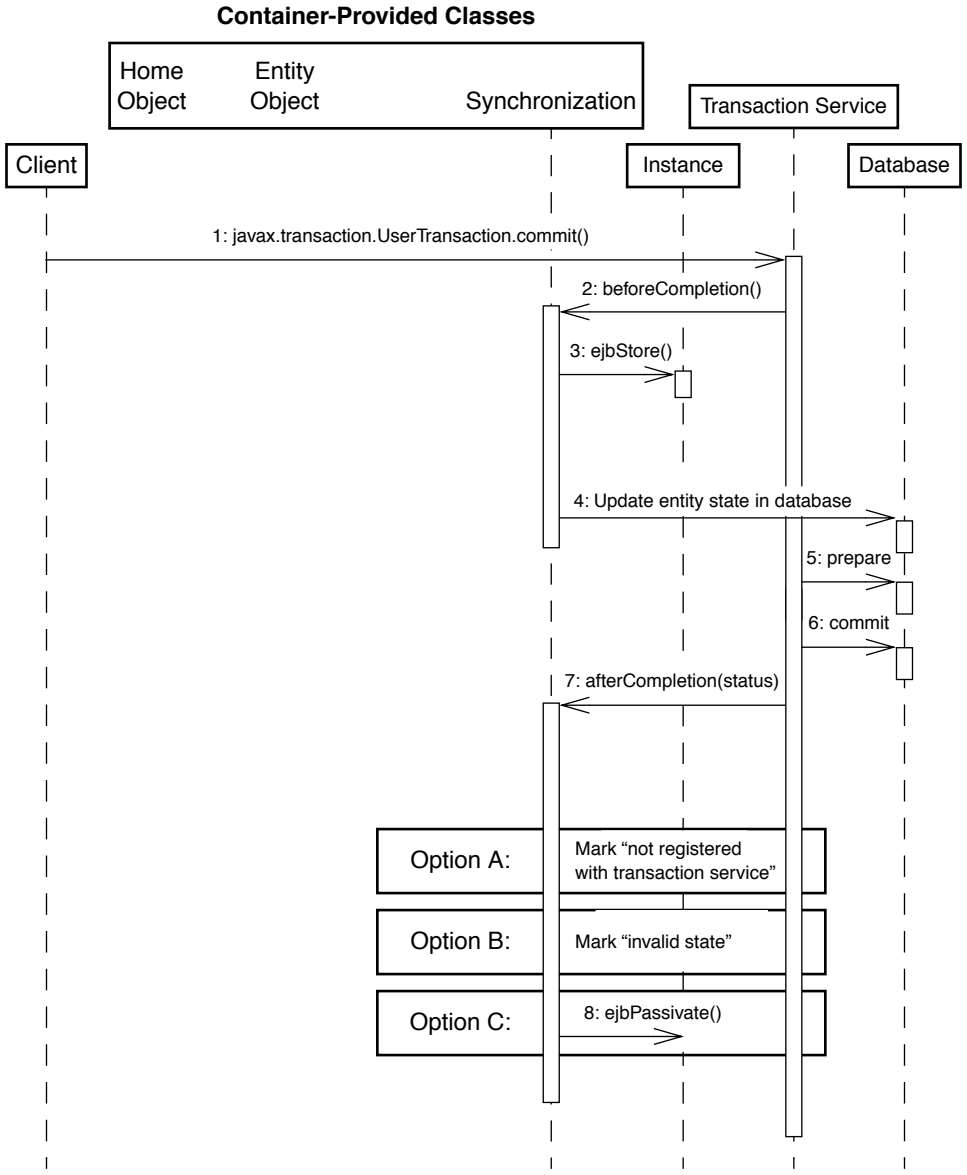

Figure 7.16 OID of Transaction Commit Protocol for CMP Entity Instance

Start of Next Transaction OID

This section describes the operations that occur at the start of the next transaction. Figure 7.17 shows the object interactions at the start of the next transaction for an entity instance with BMP.

1. The client starts the next transaction by invoking the `begin` method on the `UserTransaction` interface.
2. The client then invokes a business method on the entity object that implements the remote interface.
3. If it uses commit option C, the container allocates an instance from the pool and invokes the `ejbActivate` method on it.
4. If it uses commit option B or C, the container invokes the `ejbLoad` method on the instance.
5. The instance uses the `ejbLoad` method to read the object state, or parts of the object state, from the database. (This happens only if the container uses commit option B or C.)
6. The database system registers itself with the transaction service. (This happens only if the container uses commit option B or C.)
7. The entity object creates a `Synchronization` object.
8. The entity object registers the `Synchronization` object with the transaction service.
9. The entity object delegates the invocation of the business method to the instance.
10. The client invokes the next business method in the same transaction.
11. The entity object delegates the invocation to the instance.

Figure 7.18 shows the object interactions at the start of the next transaction for an entity instance with CMP.

1. The client starts the next transaction by invoking the `begin` method on the `UserTransaction` interface.
2. The client then invokes a business method on the entity object that implements the remote interface.

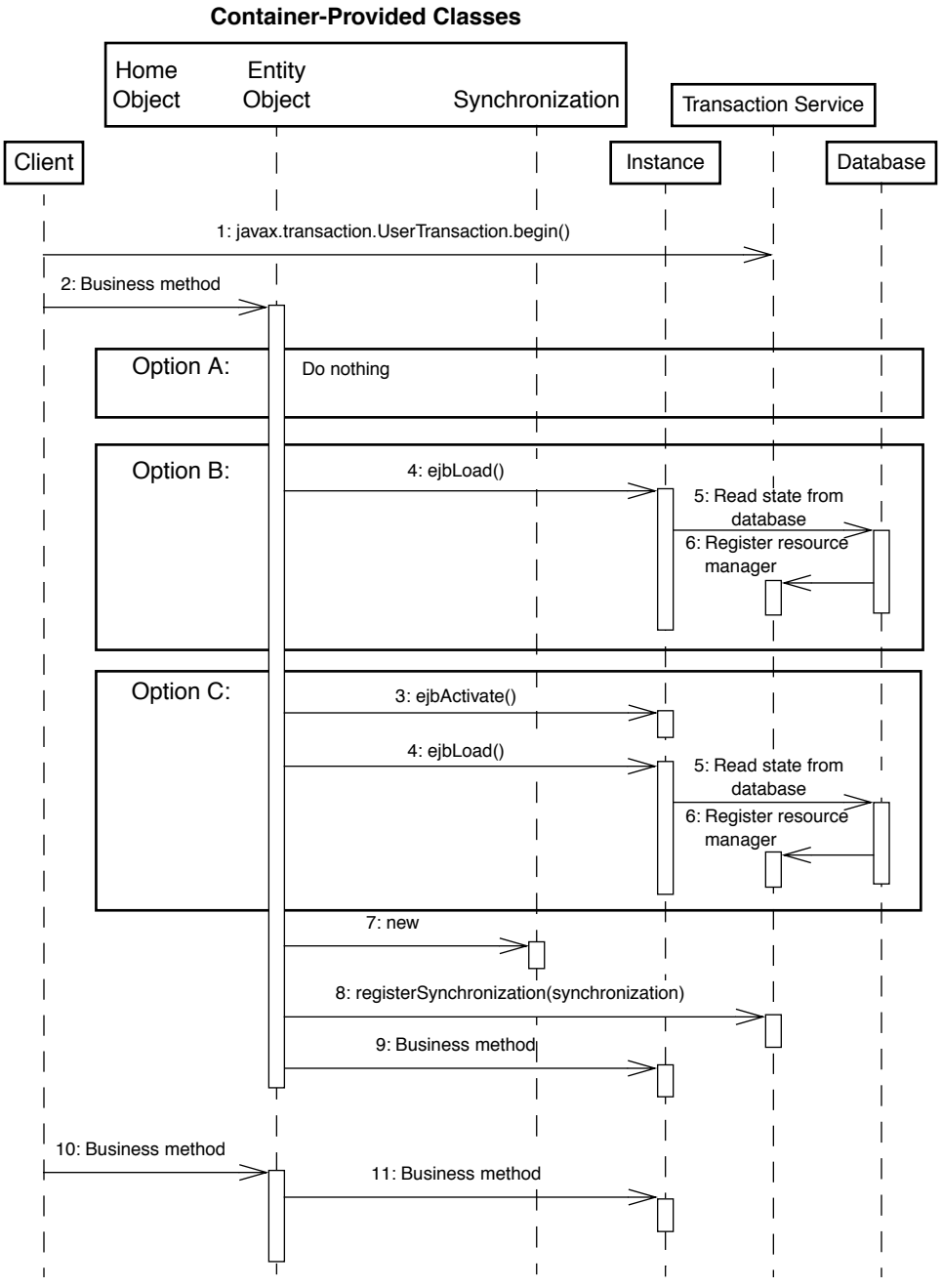

Figure 7.17 OID of Next Transaction for a Bean-Managed Entity Instance

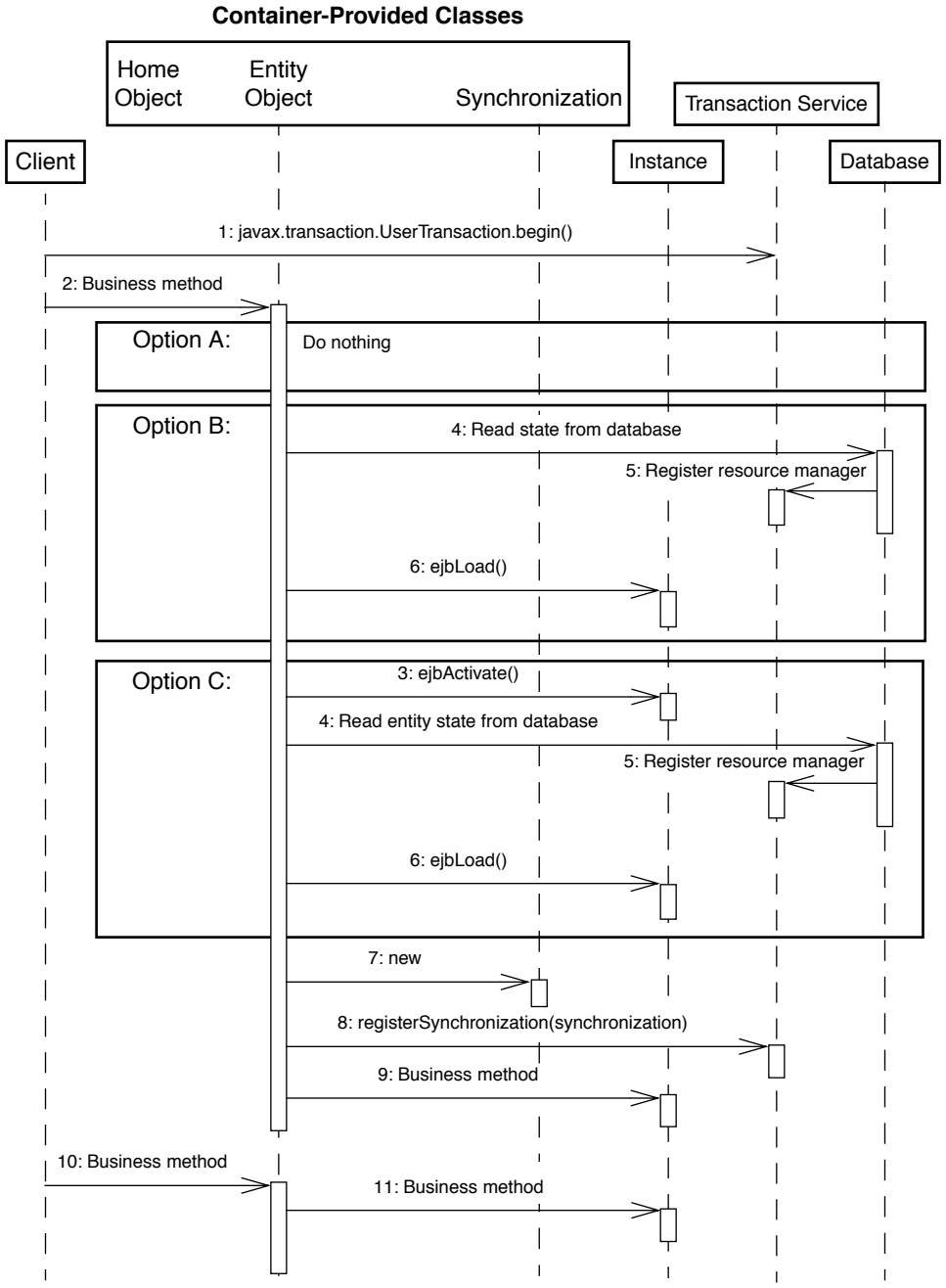

Figure 7.18 OID of Next Transaction for a Container-Managed Entity Instance

3. If it uses commit option C, the container allocates an instance from the pool and invokes the `ejbActivate` method on it.
4. If it uses commit option B or C, the container reads the values of the container-managed fields from the database.
5. The database system registers itself with the transaction service. (This happens only if the container uses commit option B or C.)
6. The container invokes the `ejbLoad` method on the instance. The implementation of the `ejbLoad` method is typically empty for instances with CMP. (This happens only if the container uses commit option B or C.)
7. The entity object creates a `Synchronization` object.
8. The entity object registers the `Synchronization` object with the transaction service.
9. The entity object delegates the invocation of the business method to the instance.
10. The client invokes the next business method in the same transaction.
11. The entity object delegates the invocation to the instance.

Commit Options

The entity bean protocol gives the container the flexibility to select the disposition of the instance state at transaction commit. This flexibility allows the container to manage optimally the caching of the entity object's state and the association of an entity object identity with the enterprise bean instances.

The container selects from the following commit options:

- **Option A**—The container caches a "ready" instance between transactions. The container ensures that the instance has exclusive access to the state of the object in the persistent storage. Because of this exclusive access, the container does not need to synchronize the instance's state from the persistent storage at the beginning of the next transaction.
- **Option B**—The container caches a "ready" instance between transactions. Unlike in option A, the container does not ensure that the instance has exclusive access to the state of the object in the persistent storage. Therefore, the container must synchronize the instance's state from the persistent storage at

the beginning of the next transaction by invoking the `ejbLoad` method on the instance.

- **Option C**—The container does not cache a "ready" instance between transactions. The container returns the instance to the pool of available instances after a transaction has completed. When the entity object is reinvoked in the next transaction, the container must activate an instance from the pool to handle the invocation.

Table 7.1 summarizes the commit options. As you can see, for all three options, the container synchronizes the instance's state with the persistent storage at transaction commit.

Table 7.1 Summary of Commit Options

Option	Write Instance State to Database	Instance Stays Ready	Instance State Remains Valid
A	Yes	Yes	Yes
B	Yes	Yes	No
C	Yes	No	No

A container can implement some or all of the three commit options. If the container implements more than one option, the deployer can typically specify which option will be used for each entity bean. The optimal option depends on the expected workload.

- Given a low probability that a client will access an entity object again, using option C will result in returning the instance to the pooled state as quickly as possible. The container can immediately reuse the instance for other object identities rather than allocating new instances.
- Given a high probability that a client may access an entity object again, using option B will result in retaining the instance associated with an object identity in the ready state. Retaining the instance in the ready state saves the `ejbPassivate` and `ejbActivate` transitions on each client transaction to the same entity object.
- Option A can be used instead of option B to improve performance further by skipping the `ejbLoad` synchronization call on the next transaction. Note that option A can be used only if it can be guaranteed that no other program can modify the underlying state in the database.

The selection of the commit option is transparent to the entity bean implementation. The entity bean works correctly regardless of the commit option chosen by the container. The bean developer writes the entity bean in the same way.

The object interaction in Transaction-Commit OID on page 210 and Start of Next Transaction OID on page 214 illustrate the commit options in detail.

7.2.4 Caching Entity Bean State

In this section, we explain how an entity bean can best use the `ejbLoad` and `ejbStore` methods in the entity bean class implementation. The container invokes the `ejbLoad` and `ejbStore` methods on the instances of both BMP and CMP entity beans. However, using the `ejbLoad` and `ejbStore` methods differs between BMP and CMP entity beans.

Caching State with BMP

Recall from earlier in this chapter that the state of an entity object is kept in a resource manager. Typically, the resource manager resides on a network node different from the EJB container in which the entity bean accessing the state is deployed. Because the implementation of a business method typically accesses the entity object's state, each invocation of a business method normally results in a network trip to the resource manager. If a transaction includes multiple business method invocations, the resulting multiple calls to the resource manager over the network may increase the transaction overhead.

Figure 7.19 shows the OID for a transaction with three calls to entity bean business methods. The business methods either read or update the entity object's state stored in the resource manager. Together with the data commit at the end of the transaction, this one transaction includes a total of four network calls to the resource manager.

1. The client starts a transaction by invoking the `begin` method, which initiates the appropriate actions from the transaction manager to create a new transaction.
2. The client, working within the transaction context, invokes the `getBalance` business method on the `Account` remote interface.
3. `AccountObject`, which implements the `Account` remote interface, in turn invokes the `ejbLoad` method on the `AccountBean` instance. In our example, the instance does no work in the `ejbLoad` method.

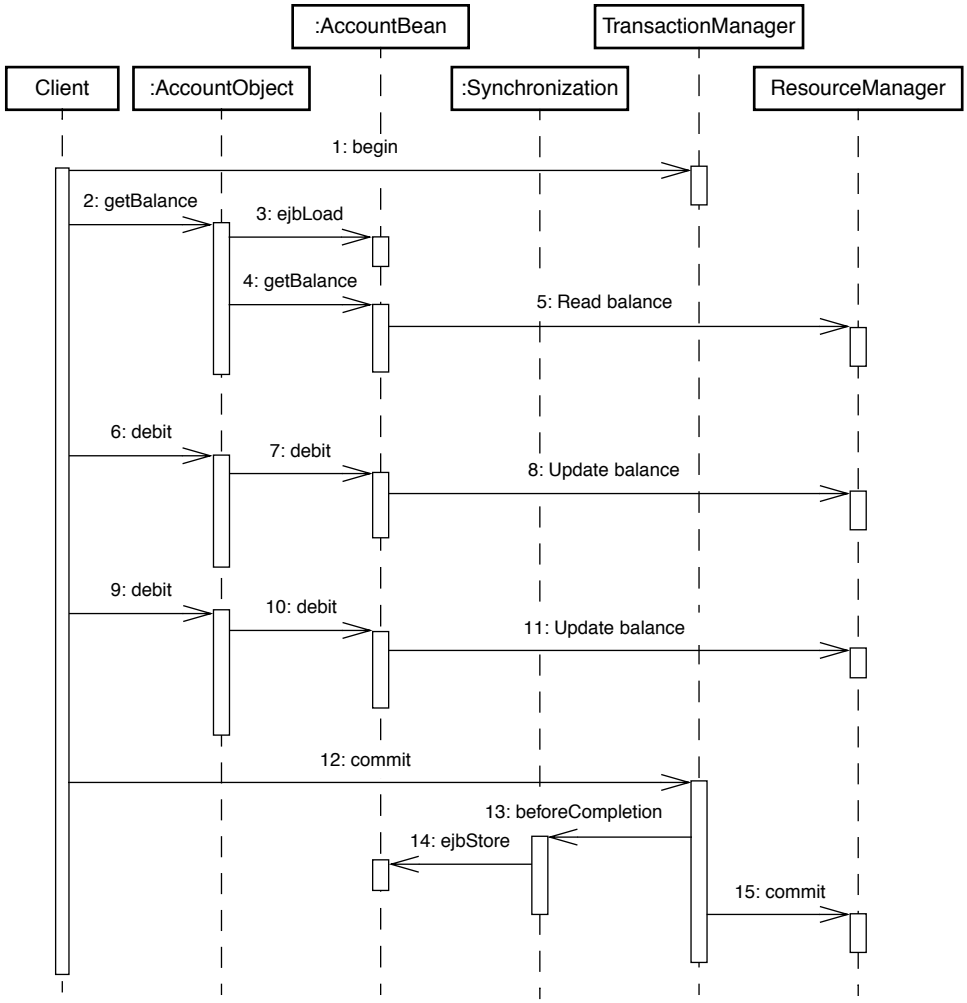

Figure 7.19 Multiple Invocations to the Resource Manager

4. `AccountObject` invokes the `getBalance` method on the `AccountBean` instance that corresponds to the `getBalance` method invoked by the client through the `Account` remote interface.

5. The `getBalance` method must access the entity object's state stored in the resource manager—that is, it must read in the account balance information stored in the database. As a result, `AccountBean` initiates a network call to access these

values from the resource manager.

6. The client invokes the `debit` method on the `Account` remote interface.
7. The `AccountObject` invokes the `debit` method on the entity bean instance.
8. The `debit` method performs the `debit` operation by updating the account balance in the resource manager. This is the second network call to the resource manager.
9. The client invokes the `debit` method a second time.
10. The previous process (steps 6 and 7) repeats: `AccountObject` invokes the `debit` method on the entity bean instance.
11. The `debit` method performs the `debit` operation by updating the account balance in the resource manager. This is the third network call to the resource manager.
12. The work of the transaction is now complete, and the client invokes the `commit` method on the transaction manager.
13. The client's invocation causes the transaction manager first to call the `beforeCompletion` method in the `Synchronization` object implemented by the container.
14. The container invokes the `ejbStore` method on the instance. In our example, the instance does no work in the `ejbStore` method.
15. The transaction manager completes the transaction by committing the results in the resource manager. This constitutes the fourth network call.

Many bean developers will want to reduce the overhead of accessing the resource manager multiple times in a transaction. To accomplish this, the EJB architecture allows the entity bean instance to cache the entity object's state, or part of its state, within a transaction. (Some containers may allow the instance to cache the entity object's state even between transactions. Such a container would use commit option A described in the section Commit Options on page 217.) Rather than making repeated calls to the resource manager to access the object's state, the instance loads the object's state from the resource manager at the beginning of a transaction and caches it in its instance variables.

To facilitate caching, the EJB container invokes the `ejbLoad` method on the instance prior to the first business method invocation in a transaction. The instance can use the `ejbLoad` method to load the entity object's state, or part of its state, into the instance's variables. Then, subsequently invoked business methods in the

instance can read and update the cached state instead of making calls to the resource manager. When the transaction ends, the EJB container invokes the `ejbStore` method on the instance. If the previously invoked business methods updated the state cached in the instance variables, the instance uses the `ejbStore` method to synchronize the entity object's state in the resource manager with the cached state. Note that `AccountBean` in Figure 7.19 did not take advantage of the `ejbLoad` and `ejbStore` methods, although the methods were called by the container.

Figure 7.20 shows the OID diagram illustrating the use of cached state for the same account balance and debit transactions in Figure 7.19.

1. The client starts a transaction by invoking the `begin` method. This initiates the appropriate actions from the transaction manager to create a new transaction.
2. The client, working within the transaction context, invokes the `getBalance` business method on the `Account` remote interface.
3. Before any business methods execute, `AccountObject` invokes the `ejbLoad` method on the `AccountBean` instance. Recall that the `AccountObject` class was generated by the container tools.
4. From the `ejbLoad` method, `AccountBean` accesses the resource manager and reads the object's state into its instance's variables.
5. `AccountObject` then invokes the corresponding `getBalance` method on the `AccountBean` instance.
6. The client does the first invocation of the `debit` method on the `Account` remote interface.
7. `AccountObject` invokes the `debit` method on the `AccountBean` instance.
8. The client does the second invocation of the `debit` method on the `Account` remote interface.
9. `AccountObject` invokes the `debit` method on the `AccountBean` class.
10. The work of the transaction completes, and the client invokes the `commit` method on the transaction manager.
11. The transaction manager invokes the `beforeCompletion` method to signal the container (via the `Synchronization` interface) that the transaction is starting its commit process.
12. The container invokes the `ejbStore` method on the `AccountBean` instance so that it properly synchronizes the object's state in the resource manager with the updated state cached in the instance variables.

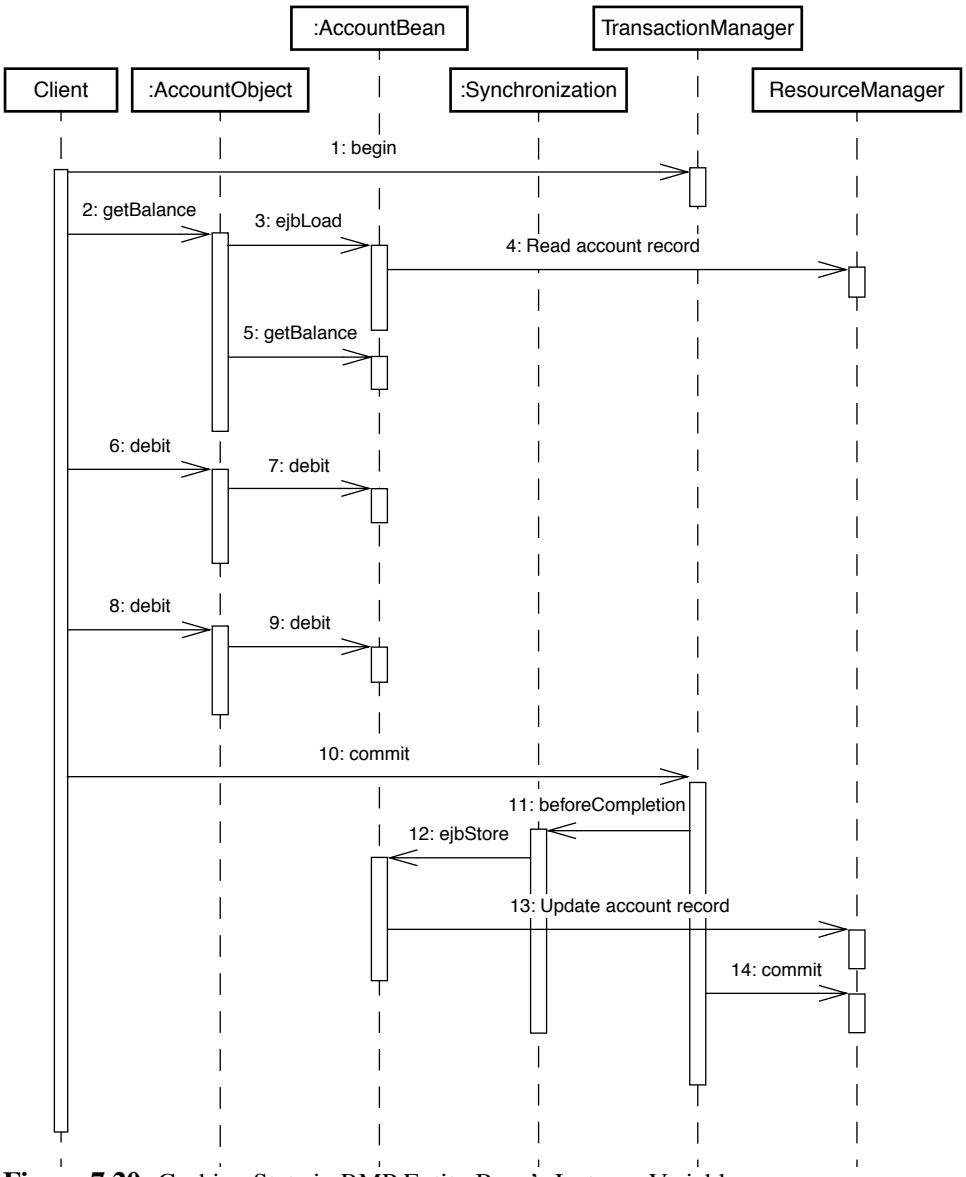

Figure 7.20 Caching State in BMP Entity Bean's Instance Variables

13. `AccountBean` sends changed state cached in its instance variables to the resource manager, which updates its copy of the entity object's state.

14. The transaction manager invokes the `commit` method on the resource manager to save all state changes and to end the transaction.

The container invokes the `ejbLoad` and `ejbStore` methods—as well as the business methods between the `ejbLoad` and `ejbStore` methods—in the same transaction context. When, from these methods, the entity bean instance accesses the entity object's state in the resource manager, the resource manager properly associates all the multiple resource manager accesses with the transaction; see steps 4 and 13.

Note that the container also invokes the `ejbStore` and `ejbLoad` methods during instance passivation and activation. The OIDs in the section Passivation and Activation OID on page 206 illustrate this. Because the container needs a transaction context to drive the `ejbLoad` and `ejbStore` methods on an entity bean instance, caching of the entity object's state in the instance variable works reliably only if the entity bean methods execute in a transaction context.

The `ejbLoad` and `ejbStore` methods must be used with great caution for entity beans with methods that do not execute with a defined transaction context. (These would be entity beans with methods that use the transaction attributes `NotSupported`, `Never`, and `Supports`.) If the business methods can execute without a defined transaction context, the instance should cache only the state of immutable entity objects. For these entity beans, an instance can use the `ejbLoad` method to cache the entity object's state, but the `ejbStore` method should always be a noop.

Caching State with CMP

Caching of an entity object's state works differently with CMP. An entity bean with CMP typically does not manage the caching of the entity object's state. Instead, the entity bean relies on the container. The container performs suitable cache management when it maps the CMP and CMR fields to the data items comprising the state representation in the resource manager.

Essentially, the EJB container makes it appear that the entity object's state and relationships load into the CMP and CMR fields at the beginning of a transaction and that changes to values of the CMP and CMR fields automatically propagate to the entity object's state in the resource manager at the end of the transaction. The business methods simply access the CMP and CMR fields as if the entity object's state were maintained directly by the class rather than in the resource manager.

The container performs the loading and saving of the state transparently to the entity bean's code. The container decides the following, typically using information provided by the deployer:

- The parts of the state that it "eagerly" loads at the beginning of a transaction—just before the container invokes the `ejbLoad` call on the instance
- The parts of the state that it "lazily" reads from the resource manager, according to when the business methods need these parts

The container propagates the updates made to the container-managed state to the resource manager immediately after invoking the `ejbStore` method (Figure 7.21).

1. The client starts a transaction by invoking the `begin` method. This initiates the appropriate actions from the transaction manager to create a new transaction.
2. The client, working within the transaction context, invokes the `getBalance` business method on the `Account` component interface.
3. Before any business methods execute, `AccountObject` reads the entity object state for the resource manager. Recall that the container tools generated the `AccountObject` class.
4. `AccountObject` invokes the `ejbLoad` method on the `AccountBean` instance.
5. `AccountObject` then invokes the corresponding `getBalance` method on the `AccountBean` instance.
6. The client does the first invocation of the `debit` method on the `Account` remote interface.
7. `AccountObject` invokes the `debit` method on the `AccountBean` instance.
8. The client does the second invocation of the `debit` method on the `Account` remote interface.
9. `AccountObject` invokes the `debit` method on the `AccountBean` instance.
10. The work of the transaction completes, and the client invokes the `commit` method on the transaction manager.
11. The transaction manager invokes the `beforeCompletion` method to signal the container—via the `Synchronization` interface—that the transaction is starting its commit process.

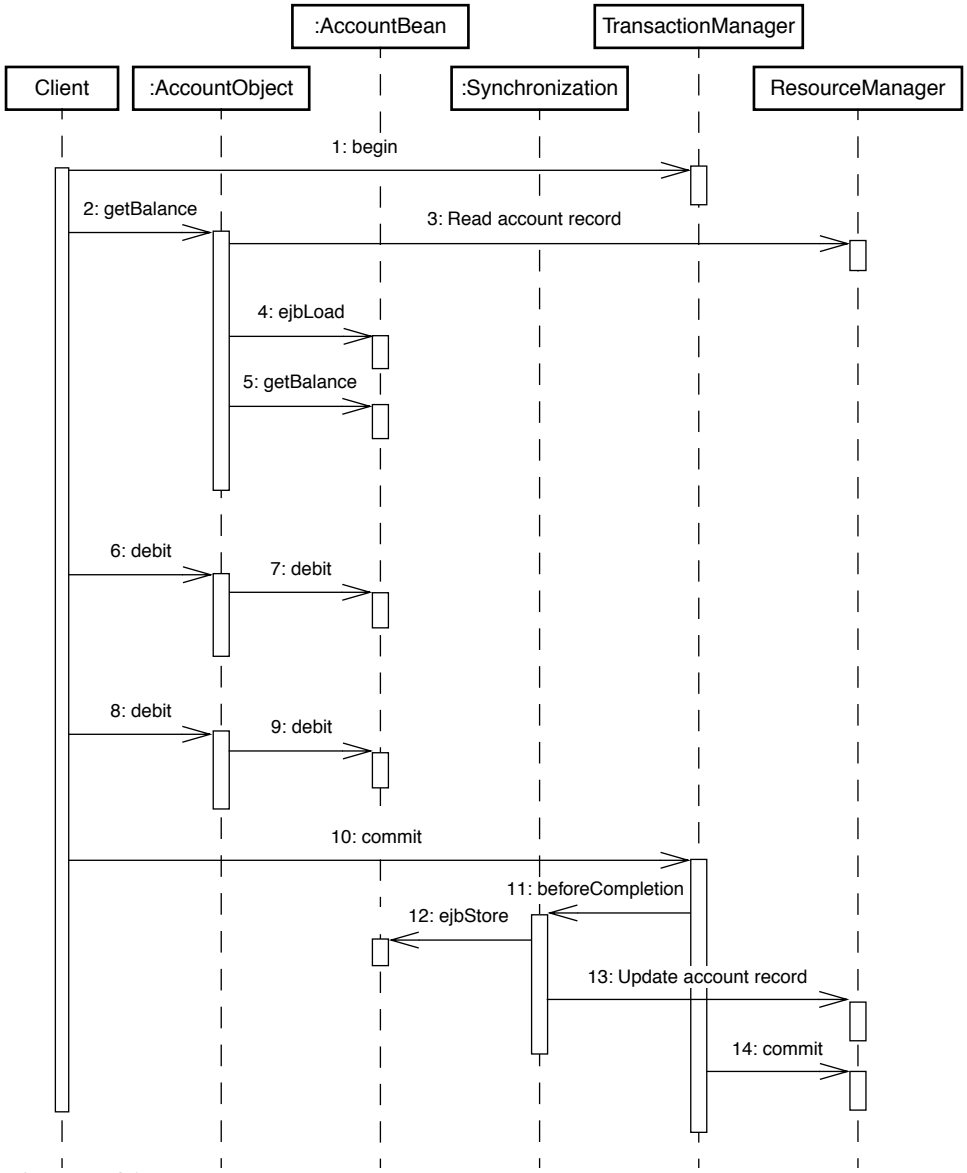

Figure 7.21 Caching with CMP Entity Beans

12. The container invokes the `ejbStore` method on the `AccountBean` instance.
13. The container updates the representation of the account state in the resource

manager with the extracted values from the CMP fields.

14. The transaction manager invokes the `commit` method on the resource manager to save all state changes and to end the transaction.

As noted at the start of this section, an entity bean with CMP typically does not use `ejbLoad` and `ejbStore` to manage caching of state. (This means that the method implementations in the entity bean class are left empty.)

When and how would an entity bean with CMP use the `ejbLoad` and `ejbStore` methods? An entity bean with CMP could use the `ejbLoad` method to compute values derived from the CMP and CMR fields. It would then use the `ejbStore` method to update the CMP and CMR fields with the updated derived values. The entity bean's business methods could then directly use the derived values. In effect, the bean instance would be caching an alternate representation of the persistent state.

Code Example 7.11 illustrates using `ejbLoad` and `ejbStore` in CMP entity beans:

```
public abstract class AccountBean implements EntityBean {
   // container-managed fields
public abstract String getAccountNumber();    // accountNumber
   public abstract void setAccountNumber(String v);

   public abstract double getBalance();  // balance in native
                                         // currency
   public abstract void setBalance(double v);

   // fields containing values derived from CMP fields
   double balanceInEuros;
   ...

   public double getBalance() {
      return balanceInEuros;
   }

   public void debit(double amountInEuros) {
      balanceInEuros = balanceInEuros - amountInEuros;
   }

   public void credit(double amountInEuros) {
```

```
            balanceInEuros = balanceInEuros + amountInEuros;
        }

        public void ejbLoad() {
            balanceInEuros = balance * conversionFactor;
        }

        public ejbStore() {
            setBalance(balanceInEuros / conversionFactor);
        }
        }
    }
```

Code Example 7.11 Using `ejbLoad` and `ejbStore` in CMP Entity Beans

`AccountBean` in Code Example 7.11 is designed for new applications that use the euro as the common currency for all currency calculations. Therefore, the methods of the `Account` component interface expect currency amounts in euros. However, the legacy account information stored in the resource manager uses the country's native currency. The CMP `AccountBean` implements the `ejbLoad` and `ejbStore` methods to perform the conversions from the native currency to euros, and vice versa.

7.2.5 Designing the Entity Bean Component Interface

The entity bean developer is responsible for the design of the bean's component interface. The entity bean developer needs to consider carefully how the bean's clients might use the methods of the component interface.

Generally, it's best to implement an entity bean with a local client view. A local client view ensures that the entity bean performs maximally and gives the bean developer the greatest latitude in designing its accessor methods. Because an entity bean with a local client view does not have the remote method invocation performance overhead, the bean developer can include individual accessor methods for each persistent attribute, regardless of the number of attributes.

If the application requires a remote view, the bean developer can use a session bean with a remote client view as a facade to entity beans with local views. This approach allows the entity bean to have the advantages of a local client view while

at the same time exposing its functionality to remote clients. An entity bean can also be implemented with both a local and a remote client view.

Entity Beans with Remote and Local Client Views

A bean developer can implement an entity bean with both a local and a remote client view. Often, such a dual-purpose entity bean has one or more entity beans with local interfaces behind it. The remote entity bean interface provides a coarse-grained view of the persistent data modeled by the network of entity beans related through their local interfaces.

Clients may directly access the entity bean's remote client view. Or, the bean developer may implement a session bean with a remote interface as a facade to the entity bean with both the local and the remote view. A client may access an entity bean method exposed through both the bean's local and remote interfaces. If this occurs, the method is called with pass-by-reference semantics in the local case—when the client is located on the same JVM as the entity bean—and with pass-by-value semantics in the remote case. To avoid unintended side effects, the bean developer should keep the parameter-passing semantic differences in mind when writing the method.

Session Bean As Facade to an Entity Bean

A bean developer can implement a session bean with a remote client view and have that session bean serve as a facade to entity beans with local views. Clients use the methods of the remote session bean, which in turn accesses the functionality of the local entity beans. The session bean implements a remote view, making the functionality of the local entity beans available to its remote clients and thus freeing its clients from being restricted to the same Java virtual machine as the session bean.

By using session beans in this way, developers can manage the interaction of clients with various entity beans. In addition, the client is exposed to a single interface, which serves as a central entry point to multiple entity beans. The client interacts exclusively with the session bean and may not even be aware of the underlying entity beans in the system. This approach is useful when an application relies on multiple entity beans, as when these beans model complex business data. For particularly complex applications, the bean developer can define different session beans to serve as facades to different functional areas of the application. Generally, however, the developer should not have a session facade for each entity bean, because such an approach is not an efficient use of server resources.

Designing Accessor Methods for Entity Beans with Remote Views

Entity beans should be designed with local views, but some situations require entity beans to use a remote client view. For these situations, we describe three different approaches to the design of the entity bean's remote interface, and discuss the trade-offs with each approach.

Accessor Methods for Individual Attributes

The entity bean developer can choose to define the remote interface methods to promote individual access to each persistent attribute. With this style, the entity bean developer defines separate accessor methods in the remote interface for each individual attribute, as shown in Code Example 7.12:

```
public interface Selection extends EJBObject {
   Employee getEmployee() throws RemoteException;
   int getCoverage() throws RemoteException;
   Plan getMedicalPlan() throws RemoteException;
   Plan getDentalPlan() throws RemoteException;
   boolean getSmokerStatus() throws RemoteException;
   void setEmployee(Employee v) throws RemoteException;
   void setCoverage(int v) throws RemoteException;
   void setMedicalPlan(Plan v) throws RemoteException;
   void setDentalPlan(Plan v) throws RemoteException;
   void setSmokerStatus(boolean v) throws RemoteException;
}
```

Code Example 7.12 Accessor Methods for Individual Attributes

This style may be useful when the entity bean may have many attributes and most clients need access to only a few attributes but the set of attributes used by each client is not known beforehand. The client retrieves only the attributes needed. However, there are drawbacks to this style:

- Each client invocation of an accessor method results in a network call, reducing the performance of the application.
- If several attributes need to be updated in a transaction, the client must use client-side transaction demarcation to make the invocation of the individual set

methods atomic. The transaction is also open across several network calls, further reducing overall performance by increasing lock contention on the underlying data and by adding the overhead of additional network trips for the transaction demarcation.

Unless the application requires the use of individual access methods, we recommend that the bean developer use one of the two other styles described next for the design of an entity bean's remote interface.

Accessing All Attributes in One Value Object

As an alternative to accessing attributes individually, the developer can define the remote interface methods to access all attributes in one call. This is a good approach when client applications typically need to access most or all of the attributes of the entity bean. The client makes one method call, which transfers all individual attributes in a single value object.

Accessing Separate Value Objects

In some cases, the developer may choose to apportion the individual attributes into subsets—the subsets can overlap—and then define methods that access each subset. This design approach is particularly useful when the entity bean has a large number of individual attributes but the typical client programs need to access only small subsets of these attributes. The entity bean developer defines multiple value objects, one for each subset of attributes. Each value object meets the needs of a client use case; each value object contains the attributes required by a client's use of that entity bean.

Defining the appropriate value objects and suitable business methods for the individual use case has two benefits. First, because the business method typically suggests the intended use case, it makes it easier for the client programmer to learn how to use the entity bean. Second, it optimizes the network traffic because the client is sent only the data that it needs, and the data is transferred in a single call. The `BankAccount` remote interface in Code Example 7.13 illustrates this style:

```
public interface BankAccount extends EJBObject {
   // Use case one
   Address getAddress()
      throws RemoteException, BankAccountException;
```

```
    void updateAddress(Address changedAddress)
       throws RemoteException, BankAccountException;

    // Use case two
    Summary getSummary()
       throws RemoteException, BankAccountException;

    // Use case three
    Collection getTransactionHistory(Date start, Date end)
       throws RemoteException, BankAccountException;
    void modifyTransaction(Transaction tran)
        throws RemoteException, BankAccountException;

    // Use case four
    void credit(double amount)
       throws RemoteException;
    void debit(double amount)
       throws RemoteException, InsufficientFundsException;
}
```

Code Example 7.13 Using Multiple Value Objects

The developer of this entity bean recognized three different client use cases for the bean, for which the bean defines the following value objects:

- One type of client uses the bean to obtain and update the address information.
- Another client type uses the entity bean for account summary information.
- A third client type uses the entity bean to view and possibly edit transaction history.

The design of the entity bean reflects this usage, defining the value objects `Address`, `Summary`, and `Transaction` for the three different client use cases. The developer has tailored the `BankAccount` remote methods for the three client use cases.

A fourth client type uses the entity bean to perform debit and credit transactions on the account. The fourth use case does not define any value objects.

7.2.6 Concurrent Invocation of an Entity Object

Recall that an entity object differs from a session object in terms of its client invocation. Only a single client can use a session object, and the client must ensure that it invokes the methods on the object serially. In contrast, multiple clients can concurrently invoke an entity object. However, each client must invoke the methods of the entity object serially.

Although the entity object appears as a shared object that the clients can invoke concurrently, the bean developer does not have to design the entity bean class to be multithreading safe. The EJB container synchronizes multiple threads' access to the entity object. The bean developer depends on the EJB container for appropriate synchronization to an entity object when multiple clients concurrently access the object.

This section explains how the EJB container dispatches methods invoked by multiple clients through the entity object's component interface to the entity bean instances so that the potentially concurrent execution of multiple client requests does not lead to the loss of the entity object's state integrity. If you are an advanced EJB developer interested in tuning your application server, you might find this section useful.

An EJB container may use one of two typical implementation strategies to synchronize concurrent access from multiple clients to an entity object. It is important to note that, from the bean developer's perspective, it makes no difference which strategy the container uses. The bean developer implements the same code for the entity object, regardless of the container synchronization strategy.

One implementation strategy essentially delegates the synchronization of multiple clients to the resource manager. It works as follows:

1. When a client-invoked method call reaches the container, the container first determines the target transaction context in which to invoke the business method. Chapter 10, Understanding Transactions, describes the rules for determining the transaction context for the invoked business method (see Section 10.1, Declarative Transaction Demarcation, on page 325).
2. The container attempts to locate an entity object instance that is in the ready state and already associated with the target entity object identity and the target transaction context. If such an instance exists, the container dispatches the business method on the instance.
3. If bean instances in the ready state are associated with the target entity object identity but none are in the target transaction context, the container can take an

instance not currently associated with a transaction context, invoke the `ejbLoad` method on the instance, and then dispatch the business method on it. The instance stays associated with the transaction context until the end of the transaction. Ending the transaction causes the container to invoke `ejbStore` on the instance; note that `ejbStore` executes in the transaction context.

4. If no bean instances in the ready state are suitable for dispatching the business method, the container activates an instance in the pooled state by invoking the `ejbActivate` and `ejbLoad` methods and then dispatches the business method on the instance. The instance remains associated with the transaction context, as described in the previous step.

Note that when it uses this implementation strategy, the container may concurrently dispatch multiple invocations to the same entity object by using a different instance of the entity bean class for each invocation (see Figure 7.22). In Figure 7.22, the two `Account 100` instances are associated with the same `Account 100` entity object, but each instance is in a different transaction context. Because the transaction context is associated with an instance's access to the resource manager, the resource manager performs the synchronization of the updates from the multiple bean instances to the entity object's state based on the transaction context. An EJB container using this implementation strategy uses commit option B or C described in the section Commit Options on page 217.

The second implementation strategy places a greater burden for the synchronization of client calls on the container. With this strategy, the EJB container acquires exclusive access to the entity object's state in the database. The container activates a single instance and serializes the access from multiple transactions to this instance (see Figure 7.23). The container can use any of the commit options A, B, or C if it uses this single-instance strategy.

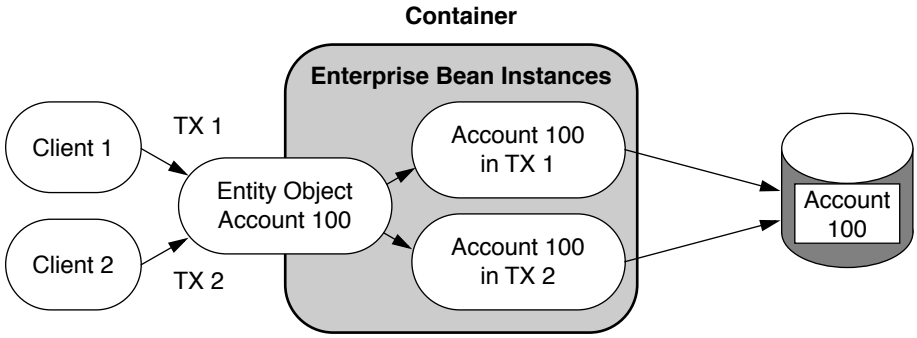

Figure 7.22 Multiple Clients Using Multiple Instances to Access an Entity Object

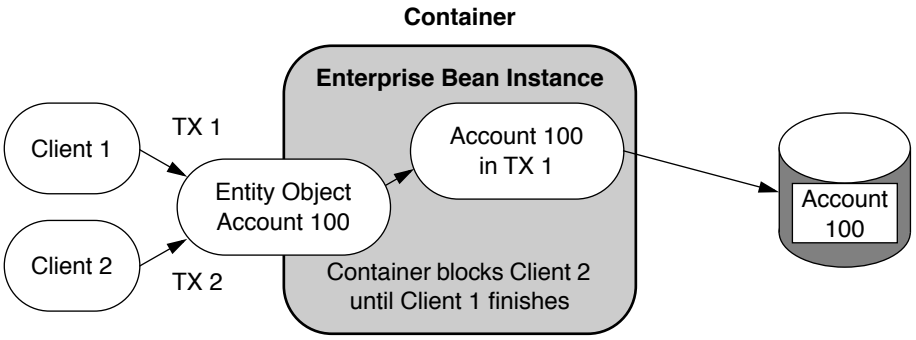

Figure 7.23 Multiple Clients Using Single Instance to Access an Entity Object

7.2.7 Using Entity Beans with Preexisting Data

It is important to understand that the representation of an entity object in the resource manager may preexist the deployment of an entity bean. For example, a bank might have been using non-object-based applications that stored account records in a database system. The bank then later developed or purchased an EJB application that included the AccountEJB entity bean, which provided the object-oriented EJB client view of the same account database records (see Figure 7.24).

The new EJB application seamlessly coexists with the legacy non-object-based application, as follows:

- An account record created by a legacy application is visible to the EJB application as an `Account` object. From the perspective of the EJB application, an `Account` object exists even if a create method of the entity bean home interface did not create it.
- Similarly, if the EJB application creates a new `Account` object, the legacy application can access the state of the entity object because it is a record in the database.
- If the EJB application changes the `Account` object by invoking methods of the `Account` remote interface, these changes are visible to a legacy application as changes to the account record in the database.

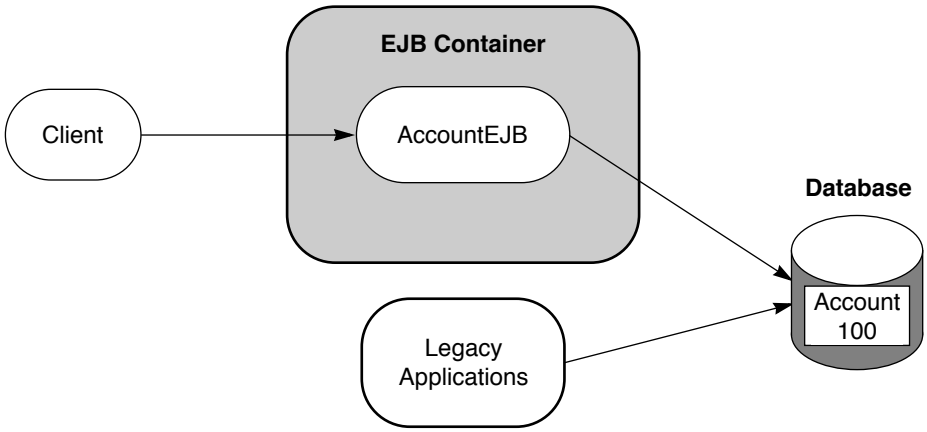

Figure 7.24 Access to Preexisting Data Shared with Legacy Applications

- Similarly, if the legacy application changes the account record in the database, the changes are visible to the EJB application as changes to the state of the `Account` object.
- If the legacy application deletes an account record from the database, the EJB application is no longer able to access the corresponding `Account` object.
- If the EJB application removes an `Account` object, the object-removal operation causes the deletion of the corresponding account record from the database. The legacy application is no longer able to access the record.
- The resource manager's—database system—transaction mechanism allows the EJB application to be used concurrently with the legacy application.

7.3 Timer Service

The EJB timer service is a new feature added to the EJB architecture in the EJB 2.1 specification. This feature can be used with both BMP and CMP entity beans, as well as with stateless session beans and message-driven beans. In Section 8.2, Parts Developed by Wombat, on page 249, we provide an example that illustrates use of the timer service with entity beans.

Business applications generally need a way to perform actions at specific times—for example, preparing a summary report of each day's transactions at the end of every business day—or after a certain interval of time—for example, if an acknowledgment for a business operation is not received in a certain time frame, alert an administrator. The EJB timer service can be used to provide such notifications to enterprise beans at a specific time or after specific time intervals. Notifications can be provided either once or on a recurring basis after a specified interval.

The EJB timer service is intended to be used for coarse-grained—hours, days, or longer—notifications that are usually needed for business processes. It is not designed to be accurate for fine-grained—milliseconds, seconds—time periods or for any kind of real-time processing.

7.3.1 Timer Interfaces

To set up a timer notification, an enterprise bean uses the EJB container's timer service to create a `Timer` object. The timer service is accessed using the `EJBContext.getTimerService` method. Code Example 7.14 shows the timer-related interfaces `TimerService`, `Timer`, `TimerHandle`, and `TimedObject`:

```
public interface TimerService {

    public Timer createTimer(long duration, Serializable info)
        throws java.lang.IllegalArgumentException,
        java.lang.IllegalStateException, javax.ejb.EJBException;

    public Timer createTimer(long initialDuration,
        long intervalDuration, Serializable info) throws
            java.lang.IllegalArgumentException,
            java.lang.IllegalStateException, javax.ejb.EJBException;

    public Timer createTimer(Date expiration, Serializable info)
        throws java.lang.IllegalArgumentException,
        java.lang.IllegalStateException, javax.ejb.EJBException;

    public Timer createTimer(Date initialExpiration,
        long intervalDuration, Serializable info) throws
        java.lang.IllegalArgumentException,
        java.lang.IllegalStateException, javax.ejb.EJBException;
```

```
    public Collection getTimers() throws
        java.lang.IllegalStateException, javax.ejb.EJBException;
}

public interface Timer {

    public void cancel() throws java.lang.IllegalStateException,
        javax.ejb.NoSuchObjectLocalException,
        javax.ejb.EJBException;

    public long getTimeRemaining() throws
        java.lang.IllegalStateException,
        javax.ejb.NoSuchObjectLocalException,
        javax.ejb.EJBException;

    public Date getNextTimeout() throws
        java.lang.IllegalStateException,
        javax.ejb.NoSuchObjectLocalException,
        javax.ejb.EJBException;

    public Serializable getInfo() throws
        java.lang.IllegalStateException,
        javax.ejb.NoSuchObjectLocalException,
        javax.ejb.EJBException;

    public TimerHandle getHandle() throws
        java.lang.IllegalStateException,
        javax.ejb.NoSuchObjectLocalException,
        javax.ejb.EJBException;
}

public interface TimerHandle extends Serializable {

    public Timer getTimer() throws java.lang.IllegalStateException,
        javax.ejb.NoSuchObjectLocalException,
        javax.ejb.EJBException;
}

public interface TimedObject {
```

```
    public void ejbTimeout(Timer timer);
}
```

Code Example 7.14 Timer Interfaces

EJB application code uses the `createTimer` methods of `TimerService` to create `Timer` objects. There are four such methods:

- Two methods create timers that expire after a specified duration. This can be a one-time timer or a recurring timer with a specified interval between notifications.
- Two methods create timers that expire at a specified time. This, too, can be a one-time timer or a recurring timer.

A bean can create multiple timer objects. The bean can associate an information object with each timer object to help the bean differentiate between multiple timer objects.

When a timer expires, the container sends a notification to the bean that created the timer, by calling the timer's `ejbTimeout` method and providing the timer as an argument. This requires that all beans that create timers must implement the `TimedObject` interface.

Timer objects implement the `Timer` interface; which provides methods to cancel—delete—the timer; get the time remaining before the timer expires; get the time when the timer will expire; get the information object associated with the timer; and get the timer's `TimerHandle` object. A bean can get a reference to a `Timer` object in one of three ways:

1. As a return value of the `createTimer` methods on the `TimerService` interface
2. As the parameter of the `ejbTimeout` method implemented by the bean class
3. From the `TimerHandle` object for a previously created timer

The `TimerHandle` object provides a serializable handle to a `Timer` object. A bean can store this handle in a database or other persistent storage and later retrieve it. Entity beans using container-managed persistence may store the `TimerHandle` object in a CMP field.

7.3.2 Timers, Persistence, and Transactions

Timer objects are persistent objects managed by the container. This means that a timer object survives crashes of the container or process in which the enterprise bean created the timer. Typically, a container implementation would store timer objects in a database or other persistent storage, so the timer's notifications could be delivered and so the timer object could be accessed even after a container restart.

For entity beans, a timer object is associated with the bean instance—identified by a primary key—that created the timer. Thus, when a container process restarts after a crash, the container first activates an instance of that entity bean with the primary key and then, if there are any outstanding expired timer notifications for the bean, delivers these timer notifications to the bean by calling its `ejbTimeout` method. If an entity bean instance is removed, all timers associated with that bean instance are also removed.

Because there is no identity associated with instances of stateless session beans and message-driven beans, to deliver the timer notification, the container uses any bean instance of the same type as the stateless session or message-driven bean that created the timer.

Timer objects are also transactional objects. This means that timer creation, removal, and expiration typically happen within the context of a transaction. If that transaction rolls back, the timer operation is also rolled back. This results in the following semantics of timers and transactions:

- If a timer is created within a transaction and the transaction rolls back, the timer creation is rolled back—as if the timer were never created.
- If a previously created timer is deleted in a transaction and the transaction rolls back, the timer deletion is rolled back—as if the timer were never deleted. Timer notifications continue to be delivered.
- If a bean's `ejbTimeout` method has the transaction attribute `RequiresNew`, the container starts a transaction before delivering a timer notification. If that new transaction rolls back, the timer notification is also rolled back, and the container will redeliver the timer notification. Note that the `ejbTimeout` method's transaction attribute is restricted to either `RequiresNew` or `NotSupported`.

7.4 Conclusion

This chapter presented the fundamental concepts of entity beans. In particular, it described the two views of entity beans: the client view—both remote and local views—and the bean developer view. The chapter described the techniques for managing the persistent state of entity beans: container-managed persistence and the EJB QL language, and bean-managed persistence. The chapter also described how the container manages the life cycle of an entity bean instance and how the bean developer can make optimal use of the container's management of object state and persistence. Finally, it described the EJB timer service and how to use it with entity beans.

The next chapter presents the same Benefits Enrollment example that was used earlier to illustrate session beans but this time is used as an application that uses entity beans. The example application illustrates all the concepts discussed in this chapter with "real-world" code. The application illustrates many of the techniques for using entity beans to develop and integrate applications for different customers with different environments.