

Oracle Application Server TopLink Unit of Work Primer

*An Oracle White Paper
August 2003*

OracleAS TopLink Unit of Work Primer

<u>INTRODUCTION</u>	3
<u><i>Example Object Model and Data Model</i></u>	4
<u>ARCHITECTURE</u>	6
<u><i>Lifecycle</i></u>	6
<u>BASIC API</u>	7
<u><i>Acquiring a Unit of Work</i></u>	7
<u><i>Creating an Object</i></u>	7
<u><i>Modifying an Object</i></u>	9
<u><i>Associations: New Target to Existing Source Object</i></u>	9
<u><i>Associations: New Source to Existing Target Object</i></u>	11
<u><i>Associations: Existing Source to Existing Target Object</i></u>	12
<u><i>Deleting Objects</i></u>	13
<u>ADVANCED API</u>	15
<u><i>Troubleshooting a Unit of Work</i></u>	15
<u><i>Creating and Registering an Object In One Step</i></u>	18
<u><i>Using registerNewObject</i></u>	18
<u><i>Using registerAllObjects</i></u>	21
<u><i>Using Registration and Existence Checking</i></u>	21
<u><i>Working with Aggregates</i></u>	22
<u><i>Unregistering Working Clones</i></u>	22
<u><i>Declaring Read-Only Classes</i></u>	23
<u><i>Conforming Queries</i></u>	23
<u><i>Using commitAndResume</i></u>	25
<u><i>Using a Nested Unit of Work</i></u>	25
<u><i>Using a Unit of Work with Custom SQL</i></u>	26
<u><i>Using a Unit of Work with JTA</i></u>	26
<u><i>Controlling the Order of Deletes</i></u>	30
<u>CONCLUSION</u>	32
<u>BEST PRACTICES</u>	33
<u>GLOSSARY</u>	34

OracleAS TopLink Unit of Work Primer

INTRODUCTION

Oracle Application Server TopLink (TopLink) is an advanced object-to-relational persistence architecture, suitable for a wide range of Java 2 Enterprise Edition (J2EE) and Java applications that store persistent data in a relational database.

TopLink changes persistent data using a transaction pattern. A transaction is a set of operations (adds, deletes, and changes) that either succeeds or fails as a single unit. If any one operation fails, no changes are made to the underlying relational database or the persistent objects or beans stored in the shared cache.

Transactions in TopLink are implemented by the Unit of Work object. Using the Unit of Work, you can transactionally modify objects directly or by way of a J2EE external transaction controller (such as JTA).

Overview

This white paper provides the background information and examples essential to understanding and effectively using the TopLink Unit of Work.

It describes:

- Principle components of a Unit of Work and how they are used (see "Architecture")
- Essential API calls most commonly used throughout the development cycle (see "Basic API")
- Specialized API calls used less frequently and usually during performance enhancement and optimization cycles (see "Advanced API")

The Unit of Work is used within TopLink's persistence manager solutions for Container Managed Persistence (CMP) of EJB Entity Beans as well as being the recommended solution for applying changes to persistent Java Objects. This document focuses on the direct use of the Unit of Work and other related TopLink API. Although the information may be useful to TopLink users developing with CMP the code samples will not be as the interaction with TopLink is well encapsulated with the J2EE container's integration.

For a list of recommendations and tips drawn from throughout this white paper, see "Best Practices".

For definitions of terms used in this white paper, see the "Glossary".

Before You Begin

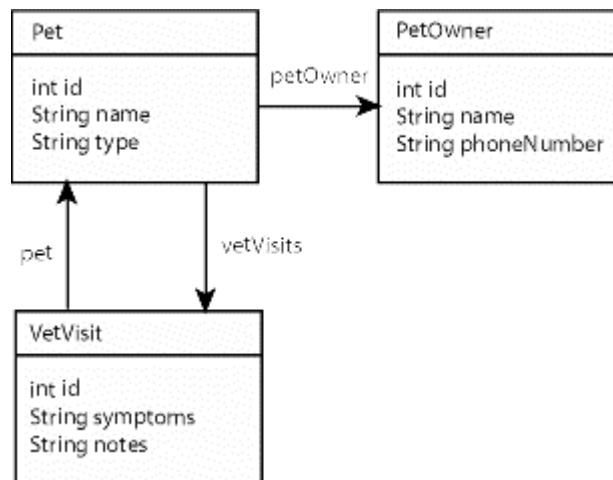
Before proceeding, familiarize yourself with the example object model and database schema described in "Example Object Model and Schema". Reference is made to these throughout this white paper.

Example Object Model and Data Model

The examples in this white paper are based on the object model and data model (entity-relationship) described here.

Object Model

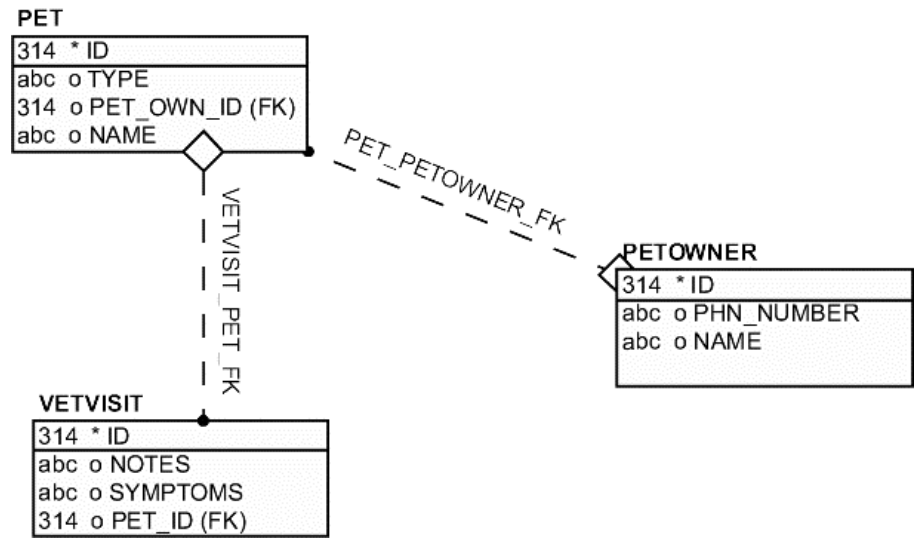
The object model is shown in the figure below.



Note that in this example object model, Pet is the source with respect to PetOwner (which is the target) and Pet is also the source with respect to VetVisit (which is also the target). VetVisit has a back-reference to Pet.

Data Model

The example data model is shown in the figure below:



Architecture

The Unit of Work executes changes on copies, or clones, of objects and if successful, applies changes to the database and Session cache. The Unit of Work is fully self-contained. Its operations occur within a Unit of Work context, isolated from the Session cache, database, and other Unit of Work transactions.

Lifecycle

The Unit of Work in an application is typically used as follows:

1. Client application acquires a Unit of Work from a Client Session object.
2. Client application queries Client Session to obtain the necessary objects to modify and then registers these cached objects with the Unit of Work.

Each time an object is registered, the Unit of Work accesses the object from the Session cache or database and creates two clones:

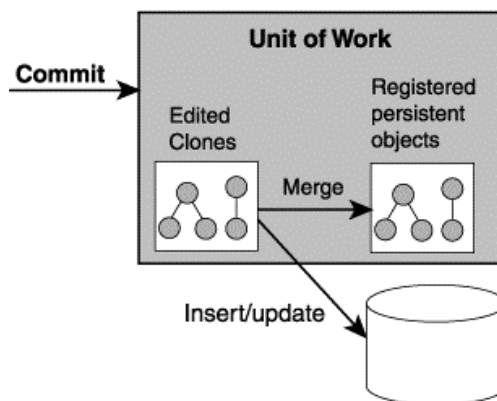
- the backup clone, and
- the working clone

As each object is registered, the Unit of Work returns the working clone to the client application.

3. Client application modifies the working clones.
4. Client application (or external transaction controller) commits transaction.

At commit time, the Unit of Work compares the working clones to the backup clones and calculates the change set (that is, determines the minimum changes required). The comparison is done with a backup clone so that concurrent changes to the same objects will not result in incorrect changes being identified. It then attempts to commit any new or changed objects to the database.

If the commit succeeds, the Unit of Work merges changes into the Server Session's cache. Otherwise, no changes are made to the objects in the shared cache.



BASIC API

The following examples explore the essential Unit of Work API calls most commonly used throughout the development cycle:

- "Acquiring a Unit of Work"
- "Creating an Object"
- "Modifying an Object"
- "Associations: New Target to Existing Source Object"
- "Associations: New Source to Existing Target Object"
- "Associations: Existing Source to Existing Target Object"
- "Deleting Objects"

Each example highlights best practices and where appropriate, provides generated SQL, common errors, and coding style recommendations.

BASIC API

Acquiring a Unit of Work

This example examines how to acquire a Unit of Work from the Session object.

```
Server server =
    (Server) SessionManager.getManager().getSession(
        sessionName, MyServerSession.class.getClassLoader()
    );
Session session = (Session) server.acquireClientSession();
UnitOfWork uow = session.acquireUnitOfWork();
```

You can acquire a Unit of Work from any session type. Note that you do not need to create a new session and login before every transaction. The recommended pattern is to acquire a client session per client access (or thread) and then acquire the necessary unit of Work from this session.

The Unit of Work is valid until the commit (or release) method is called. After commit, a Unit of Work is not valid even if the transaction fails and is rolled back.

A Unit of Work remains valid after the commitAndResume method is called as described in "Using commitAndResume".

When using a Unit of Work with Java Transaction API (JTA)/Java Transaction Service (JTS), you can also use the advanced API on session getActiveUnitOfWork method as described in "Using a Unit of Work with JTA".

BASIC API

Creating an Object

This example examines how to create and persist a simple object (without relationships) using the clone returned by the Unit of Work registerObject method.

```
UnitOfWork uow = session.acquireUnitOfWork();
Pet pet = new Pet();
Pet petClone = (Pet)uow.registerObject(pet);
petClone.setId(100);
```

```
    petClone.setName("Fluffy");
    petClone.setType("Cat");
uow.commit();
```

A common alternative is shown below:

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet pet = new Pet();
    pet.setId(100);
    pet.setName("Fluffy");
    pet.setType("Cat");
    uow.registerObject(pet);
uow.commit();
```

Both approaches produce the following SQL:

```
INSERT INTO PET (ID, NAME, TYPE, PET_OWN_ID) VALUES (100, 'Fluffy',
'Cat', NULL)
```

The first example is preferred because it gets you into the pattern of working with clones and provides the most flexibility for future code changes. Later examples will show that working with combinations of new objects and clones can lead to confusion and unwanted results.

Common Errors

- Failure to register object with Unit of Work. **Symptom:** Object not written to database.
- Accidentally modifying original object instead of clone. **Symptom:** Fields will not be properly written to the database and the object in the shared cache's state is corrupted (not in sync with the database).
- Using a clone after the Unit of Work is committed. **Symptom:** Varies. Accessing an uninstantiated clone value holder after a Unit of Work commit will raise exceptions. A clone is a working copy used during a transaction and as soon as the transaction is complete, the clone must not be used.

Coding Style

- Name variables pointing to clones with the suffix "clone" or "WorkingCopy".
- Name Unit of Work variables with a "UOW" suffix.
- Indent code within a Unit of Work (that is, treat such code as if it was in a control structure).
- Never use a clone after committing the Unit of Work that the clone is from.

BASIC API

Modifying an Object

This example examines how to modify a simple direct-to-field attribute of an existing object.

A Pet is read prior to a Unit of Work: the variable aPet is the cache copy for that Pet. Inside of the Unit of Work, we must register the cache copy to get a working copy. We then modify the working copy and commit the Unit of Work.

```
// Read in any pet.
Pet pet = (Pet)session.readObject(Pet.class);
UnitOfWork uow = session.acquireUnitOfWork();
    Pet petClone = (Pet) uow.registerObject(pet);
        petClone.setName("Furry");
uow.commit();
```

In this example, we take advantage of the fact that you can query through a Unit of Work and get back clones, saving the registration step. However, the drawback is that we do not have a handle to the cache copy. If we wanted to do something with the updated Pet after commit, we would have to query the Session to get it (remember that after a Unit of Work is committed, its clones are invalid and must not be used).

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet petClone = (Pet) uow.readObject(Pet.class);
        petClone.setName("Furry");
uow.commit();
```

Both approaches produce the following SQL:

```
UPDATE PET SET NAME = 'Furry' WHERE (ID = 100)
```

Common Errors

- Working with the original object instead of a working copy. You must only make changes to clones if you want to persist the changes.

Coding Style

- Take care when querying through a Unit of Work. All objects read in the query are registered in the Unit of Work and therefore will be checked for changes at commit time. Rather than do a ReadAllQuery through a Unit of Work, it is better for performance to design your application to do the ReadAllQuery through a Session and then only register in a Unit of Work the objects that need to be changed.

BASIC API

Associations: New Target to Existing Source Object

This example examines how to associate a new target object with an existing source object with 1-many and 1-1 relationships.

There are two ways to associate a new object with an existing object:

- "Associating without Reference to the Cache Object"

- "Associating with Reference to the Cache Object"

Deciding which approach to use depends on whether or not your code requires a reference to the cache copy of the new object after the Unit of Work is committed and on how adaptable to change you want your code to be.

Associating without Reference to the Cache Object

The first way of associating a new target with an existing source is doing exactly as you would in Java: just create the association as shown below.

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet petClone = (Pet)uow.readObject(Pet.class);

    PetOwner petOwner = new PetOwner();
    petOwner.setId(400);
    petOwner.setName("Donald Smith");
    petOwner.setPhoneNumber("555-1212");

    VetVisit vetVisit = new VetVisit();
    vetVisit.setId(500);
    vetVisit.setNotes("Pet was shedding a lot.");
    vetVisit.setSymptoms("Pet in good health.");
    vetVisit.setPet(petClone);

    petClone.setPetOwner(petOwner);
    petClone.getVetVisits().addElement(vetVisit);
uow.commit();
```

This executes the proper SQL:

```
INSERT INTO PETOWNER (ID, NAME, PHN_NBR) VALUES (400, 'Donald Smith',
'555-1212')
UPDATE PET SET PET_OWN_ID = 400 WHERE (ID = 100)
INSERT INTO VETVISIT (ID, NOTES, SYMPTOMS, PET_ID) VALUES (500, 'Pet
was shedding a lot.', 'Pet in good health.', 100)
```

When associating new objects to existing objects, the Unit of Work treats the new object as if it was a clone. That is, after the commit:

```
petOwner != session.readObject(petOwner)
```

For a more detailed discussion of this fact, see "Using registerNewObject").

Therefore, after the Unit of Work commit, the variables vetVisit and petOwner no longer point to their respective cache objects: they point at working copy clones.

If you need the cache object after the Unit of Work commit, you must query for it or create the association with a reference to the cache object (as described in "Associating with Reference to the Cache Object").

Associating with Reference to the Cache Object

The second way of associating a new target with an existing source is done with reference to the cache object, as shown below.

```

UnitOfWork uow = session.acquireUnitOfWork();
Pet petClone = (Pet)uow.readObject(Pet.class);

PetOwner petOwner = new PetOwner();
PetOwner petOwnerClone = (PetOwner)uow.registerObject(petOwner);
petOwnerClone.setId(400);
petOwnerClone.setName("Donald Smith");
petOwnerClone.setPhoneNumber("555-1212");

VetVisit vetVisit = new VetVisit();
VetVisit vetVisitClone = (VetVisit)uow.registerObject(vetVisit);
vetVisitClone.setId(500);
vetVisitClone.setNotes("Pet was shedding a lot.");
vetVisitClone.setSymptoms("Pet in good health.");
vetVisitClone.setPet(petClone);

petClone.setPetOwner(petOwnerClone);
petClone.getVetVisits().addElement(vetVisitClone);
uow.commit();

```

Now, after the Unit of Work commit:

```
petOwner == session.readObject(petOwner)
```

This means that we have a handle to the cache copy after the commit, rather than a clone.

Common Errors

- Failure to associate the new object from the existing object. **Symptom:** New object does not get written to the database.

BASIC API

Associations: New Source to Existing Target Object

This example examines how to associate a new source object with an existing target object with 1-many and 1-1 relationships.

TopLink follows all relationships of all registered objects (deeply) in a Unit of Work to calculate what is new and what has changed. We saw in the previous section that when you associate a new target with an existing source, you can choose to register the object or not. If you do not register the new object, it is still reachable from the source object (which is a clone, hence it is registered). However, when you need to associate a new source object with an existing target, you must register the new object. If you do not register the new object, then it is not reachable in the Unit of Work and TopLink will not write it to the database.

For example, imagine we want to create a new Pet and associate it with an existing PetOwner. The following code will accomplish this:

```

UnitOfWork uow = session.acquireUnitOfWork();
PetOwner existingPetOwnerClone =
    (PetOwner)uow.readObject(PetOwner.class);

Pet newPet = new Pet();
Pet newPetClone = (Pet)uow.registerObject(newPet);
newPetClone.setId(900);
newPetClone.setType("Lizzard");
newPetClone.setName("Larry");
newPetClone.setPetOwner(existingPetOwnerClone);
uow.commit();

```

This generates the proper SQL:

```
INSERT INTO PET (ID, NAME, TYPE, PET_OWN_ID) VALUES (900, 'Larry',
'Lizzard', 400)
```

In this situation, you should register the new object and work with the working copy of the new object. If you associate the new object (without registering) with the PetOwner clone it will not be written to the database. If you are in a situation where you want to associate the PetOwner clone with the new Pet object, use the advanced API registerNewObject as described in "Using registerNewObject".

Common Errors

- Failure to register the new object, or registering the new object but working with the original instead of the clone. **Symptom:** New object does not get written to the database.
- Failure to register the clone and accidentally associating the cache version of the existing object with the new object. **Symptom:** At commit time, TopLink will generate an error which states that you have associated the cache version of an object (“from a parent session”) with a clone from this Unit of Work. You must work with working copies in units of work.

BASIC API

Associations: Existing Source to Existing Target Object

This example examines how to associate an existing source object with an existing target object with 1-many and 1-1 relationships.

Associating existing objects with each other in a Unit of Work is as simple as associating objects in Java. Just remember to only work with working copies of the objects.

```
// Associate all VetVisits in the database to a Pet from the database
UnitOfWork uow = session.acquireUnitOfWork();
Pet existingPetClone = (Pet)uow.readObject(Pet.class);
Vector allVetVisitClones;
allVetVisitClones = (Vector)uow.readAllObjects(VetVisit.class);
Enumeration enum = allVetVisitClones.elements();
while(enum.hasMoreElements()) {
    VetVisit vetVisitClone = (VetVisit)enum.nextElement();
    existingPetClone.getVetVisits().addElement(vetVisitClone);
    vetVisitClone.setPet(existingPetClone);
};
uow.commit();
```

Common Errors

- The most common error when associating existing objects is failing to work with the working copies. **Symptom:** Error at commit time. If you accidentally associate a cache version of an object with a working copy you will get an error at commit time indicating that you associated an object from a parent session (the cache version) with a clone from this Unit of Work.

Deleting Objects

This example examines how to delete objects with a Unit of Work, including:

- "Using privateOwnedRelationship"
- "Explicitly Deleting from the Database"
- "Understanding the Order in which Objects are Deleted"

Using privateOwnedRelationship

Relational databases do not have garbage collection like a Java Virtual Machine (JVM) does. To delete an object in Java you just de-reference the object. To delete a row in a relational database you must explicitly delete it. Rather than tediously manage when to delete data in the relational database, use the mapping attribute `privateOwnedRelationship` to make TopLink manage the garbage collection in the relational database for you.

When you create a mapping using Java, you can use its `privateOwnedRelationship` method to tell TopLink that the referenced object is privately owned: that is, the referenced object cannot exist without the parent object. For example, to make the Pet-PetOwner relationship privately owned:

```
OneToOneMapping petOwnerMapping = new OneToOneMapping();
petOwnerMapping.setAttributeName("petOwner");
petOwnerMapping.setReferenceClass(com.top.uowprimer.model.PetOwner.class);
petOwnerMapping.dontUseIndirection();
petOwnerMapping.privateOwnedRelationship();
petOwnerMapping.addForeignKeyFieldName("PET.PET_OWN_ID", "PETOWNER.ID");
descriptor.addMapping(petOwnerMapping);
```

When you create a mapping using the Mapping Workbench, you can select the **Private Owned** check box under the **General** tab.

When you tell TopLink that a relationship is private owned, you are telling it two things:

- if the source of a private owned relationship is deleted, then delete the target.
- if you de-reference a target from a source, then delete the target.

Do not configure private owned relationships to objects that might be shared. An object should not be the target in more than one relationship if it is the target in a private owned relationship.

The exception to this rule is the case when you have a many-to-many relationship in which a relation object is mapped to a relation table and is referenced through a one-to-many relationship by both the source and target. In this case, if the one-to-many mapping is configured as privately owned, then when you delete the source, all the association objects will be deleted. Here is an example using our example object model and schema:

```

// If the Pet-PetOwner relationship is privateOwned
// then the PetOwner will be deleted at uow.commit()
// otherwise, just the foreign key from PET to PETOWNER will
// be set to null. The same is true for VetVisit.
UnitOfWork uow = session.acquireUnitOfWork();
Pet petClone = (Pet)uow.readObject(Pet.class);
petClone.setPetOwner(null);
VetVisit vvClone =
    (VetVisit)petClone.getVetVisits().firstElement();
vvClone.setPet(null);
petClone.getVetVisits().removeElement(vvClone);
uow.commit();

```

If the relationships from Pet to PetOwner and from Pet to VetVisit are not private owned, this code produces the following SQL:

```

UPDATE PET SET PET_OWN_ID = NULL WHERE (ID = 150)
UPDATE VETVISIT SET PET_ID = NULL WHERE (ID = 350)

```

If the relationships from Pet to PetOwner and from Pet to VetVisit are private owned, this code produces the following SQL:

```

UPDATE PET SET PET_OWN_ID = NULL WHERE (ID = 150)
UPDATE VETVISIT SET PET_ID = NULL WHERE (ID = 350)
DELETE FROM VETVISIT WHERE (ID = 350)
DELETE FROM PETOWNER WHERE (ID = 250)

```

Explicitly Deleting from the Database

If there are cases where you have objects that will not be garbage collected through private owned relationships (especially root objects in your object model) then you can explicitly tell TopLink to delete the row representing the object using the deleteObject API. For example:

```

UnitOfWork uow = session.acquireUnitOfWork();
Pet petClone = (Pet)uow.readObject(Pet.class);
uow.deleteObject(petClone);
uow.commit();

```

The above code generates the following SQL:

```

DELETE FROM PET WHERE (ID = 100)

```

Understanding the Order in which Objects are Deleted

The Unit of Work does not track changes or the order of operations. It is intended to isolate you from having to modify your objects in the order the database requires.

By default, at commit time, the Unit of Work orders all inserts and updates using the constraints defined by your schema. After all inserts and updates are done, the Unit of Work will issue the necessary delete operations.

Constraints are inferred from one-to-one and one-to-many mappings. If you have no such mappings, you can add additional constraint knowledge to TopLink as described in "Controlling the Order of Deletes".

ADVANCED API

The following examples explore more advanced Unit of Work API calls and techniques most commonly used later in the development cycle:

- "Troubleshooting a Unit of Work"
- "Creating and Registering an Object In One Step"
- "Using registerNewObject"
- "Using registerAllObjects"
- "Using Registration and Existence Checking"
- "Working with Aggregates"
- "Unregistering Working Clones"
- "Declaring Read-Only Classes"
- "Conforming Queries"
- "Using commitAndResume"
- "Using a Nested Unit of Work"
- "Using a Unit of Work with Custom SQL"
- "Using a Unit of Work with JTA"
- "Controlling the Order of Deletes"

Each example highlights best practices and where appropriate, provides generated SQL, common errors, and coding style recommendations.

ADVANCED API

Troubleshooting a Unit of Work

This example examines common Unit of Work problems and debugging techniques:

- "Avoiding the Use of Post-commit Clones"
- "Determining Whether or not an Object is the Cache Object"
- "Dumping the Contents of a Unit of Work"
- "Handling Exceptions"

Avoiding the Use of Post-commit Clones

A common Unit of Work error is holding on to clones after commit. Typically the clones are stored in a static variable and the developer incorrectly thinks that this object is the cache copy. This leads to problems when another Unit of Work makes changes to the object and what the developer thinks is the cache copy is not updated (because a Unit of Work only updates the cache copy, not old clones).

Consider the error in the following example. In this example we get a handle to the cache copy of a Pet and store it in the static CACHE_PET. We get a handle to a working copy and store it in the static CLONE_PET. In a future Unit of Work, the Pet is changed.

Developers who incorrectly store global references to clones from units of work often expect them to be updated when the cache object is changed in a future Unit of Work. Only the cache copy is updated.

```
//Read a Pet from the database, store in static
CACHE_PET = (Pet)session.readObject(Pet.class);

//Put a clone in a static. This is a bad idea and is a common error
UnitOfWork uow = session.acquireUnitOfWork();
    CLONE_PET = (Pet)uow.readObject(Pet.class);
    CLONE_PET.setName("Hairy");
uow.commit();

//Later, the pet is changed again
UnitOfWork anotherUow = session.acquireUnitOfWork();
    Pet petClone = (Pet)anotherUow.registerObject(CACHE_PET);
    petClone.setName("Fuzzy");
anotherUow.commit();

// If you incorrectly stored the clone in a static and thought it should be
// updated when it's later changed, you would be wrong: only the cache copy is
// updated; NOT OLD CLONES.
System.out.println("CACHE_PET is" + CACHE_PET);
System.out.println("CLONE_PET is" + CLONE_PET);
```

The two System.out calls produce the following output:

```
CACHE_PET isPet type Cat named Fuzzy id:100
CLONE_PET isPet type Cat named Hairy id:100
```

Determining Whether or not an Object is the Cache Object

In "Modifying an Object", we noted that it is possible to read any particular instance of a class by executing:

```
session.readObject(Class);
```

There is also a readObject method that takes an object as an argument: this method is equivalent to doing a ReadObjectQuery on the primary key of the object passed in. For example, the following:

```
session.readObject(pet);
```

Is equivalent to the following:

```
ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Pet.class);
ExpressionBuilder builder = new ExpressionBuilder();
Expression exp = builder.get("id").equal(pet.getId());
query.setSelectionCriteria(exp);
session.executeQuery(query);
```

Also note that primary key based queries, by default, will return what is in the cache without going to the database.

Given this, we have a very quick and simple method for accessing the cache copy of an object as shown in the following example:

```
//Here is a test to see if an object is the cache copy
boolean cached = CACHE_PET == session.readObject(CACHE_PET);
boolean cloned = CLONE_PET == session.readObject(CLONE_PET);
System.out.println("Is CACHE_PET the Cache copy of the object: " + cached);
System.out.println("Is CLONE_PET the Cache copy of the object: " + cloned);
```

This code produces the following output:

```
Is CACHE_PET the Cache copy of the object: true
Is CLONE_PET the Cache copy of the object: false
```

Dumping the Contents of a Unit of Work

The Unit of Work has several debugging methods to help you analyze performance or track down problems with your code. The most useful is `printRegisteredObjects` which prints all the information about objects known in the Unit of Work. Use this method to see how many objects are registered and to make sure objects you are working on are registered.

To use this method, you must have log messages enabled for the Session that the Unit of Work is from. Session log messages are disabled by default. To enable log messages, use the `Session.logMessages` method. To disable log messages, use the `Session.dontLogMessages` method (see below for an example.)

Consider the following example:

```
session.logMessages(); // enable log messages
UnitOfWork uow = session.acquireUnitOfWork();
    Pet petClone = (Pet)uow.readObject(Pet.class);
    petClone.setName("Mop Top");

    Pet pet2 = new Pet();
    pet2.setId(200);
    pet2.setName("Sparky");
    pet2.setType("Dog");
    uow.registerObject(pet2);

    uow.printRegisteredObjects();
uow.commit();
session.dontLogMessages(); // disable log messages
```

This example produces the following output:

```
UnitOfWork identity hashCode: 32373
Deleted Objects:

All Registered Clones:
    Key: [100] Identity Hash Code:13901  Object: Pet type Cat named Mop Top id:100
    Key: [200] Identity Hash Code:16010  Object: Pet type Dog named Sparky id:200

New Objects:
    Key: [200] Identity Hash Code:16010  Object: Pet type Dog named Sparky id:200
```

Handling Exceptions

TopLink exceptions are instances of `RuntimeException`, which means that methods that throw them do not have to be placed in a try-catch statement.

However, the Unit of Work commit method is one that should be called within a try-catch statement to deal with problems that may arise.

The following example shows one way to handle Unit of Work exceptions.

```
UnitOfWork uow = session.acquireUnitOfWork();
Pet petClone = (Pet)uow.registerObject(newPet);
petClone.setName("Assume this name is too long for a database
constraint");
// Assume that the name argument violates a length constraint on the
database.
// This will cause a DatabaseException on commit.
try {
    uow.commit();
} catch (TopLinkException tle) {
    System.out.println("There was an exception: " + tle);
}
```

This code produces the following output:

```
There was an exception: EXCEPTION [ORACLEAS TOPLINK-6004]:
oracle.toplink.exceptions.DatabaseException
```

Catching exceptions at commit time is mandatory if you are using optimistic locking because the exception raised is the indication that there was an optimistic locking problem. Optimistic locking allows all users to access a given object, even if it is currently in use in a transaction or Unit of Work. When the Unit of Work attempts to change the object, the database checks to ensure that the object has not changed since it was initially read by the Unit of Work. If the object has changed, the database raises an exception, and the Unit of Work rolls back the transaction.

ADVANCED API

Creating and Registering an Object In One Step

This example examines how to use the Unit of Work newInstance method to create a new Pet object, register it with the Unit of Work, and return a clone, all in one step. If you are using a factory pattern to create your objects (and specified this in the builder), the newInstance method will use the appropriate factory.

```
UnitOfWork uow = session.acquireUnitOfWork();
Pet petClone = (Pet)uow.newInstance(Pet.class);
petClone.setId(100);
petClone.setName("Fluffy");
petClone.setType("Cat");
uow.commit();
```

ADVANCED API

Using registerNewObject

This example examines how to use the advanced API method registerNewObject, including:

- "Registering a New Object with registerNewObject"
- "Associating New Objects with One Another"

Registering a New Object with registerNewObject

The registerNewObject method registers a new object as if it was a clone. At commit time, TopLink creates another instance of the object to be the cache version of that business object. Use registerNewObject in situations where:

- you do not need a handle to the cache version of the object after the commit and you do not want to work with clones of new objects
- you must pass a clone into the constructor of a new object and then need to register the new object

For example:

```
UnitOfWork uow = session.acquireUnitOfWork();
PetOwner existingPetOwnerClone =
    PetOwner)uow.readObject(PetOwner.class);

    Pet newPet = new Pet();
    newPet.setId(900);
    newPet.setType("Lizzard");
    newPet.setName("Larry");
    newPet.setPetOwner(existingPetOwnerClone);

    uow.registerNewObject(newPet);
uow.commit();
```

By using registerNewObject, the variable newPet should not be used after the Unit of Work is committed. The new object is the clone and if you need the cache version of the object, you need to query for it. If you needed a handle to the cache version of the Pet after the Unit of Work has committed, then you should use the first approach described in "Associations: New Source to Existing Target". In that example, the variable newPet is the cache version after the Unit of Work is committed. As an exercise, you can modify these two examples to verify if newPet is the cache copy or not after the Unit of Work (using the strategy described in "Troubleshooting a Unit of Work").

Associating New Objects with One Another

At commit time, TopLink can determine if an object is new or not. In "Associating a new target object with an existing source", we saw that if a new object is reachable from a clone, you do not even need to register it at all! TopLink virtually does a registerNewObject to all new objects it can reach from registered objects.

When working with new objects, remember the following rules:

- Only reachable or registered objects will be persisted.
- New objects or objects that have been registered with registerNewObject are considered to be working copies in the Unit of Work.
- If you do a registerObject with a new object as the argument, then the result is the clone and the argument to registerObject is considered the cache version.

The following is valid Unit of Work code:

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet newPet = new Pet();
    newPet.setId(150);
    newPet.setType("Horse");
    newPet.setName("Ed");

    PetOwner newPetOwner = new PetOwner();
    newPetOwner.setId(250);
    newPetOwner.setName("George");
    newPetOwner.setPhoneNumber("555-9999");

    VetVisit newVetVisit = new VetVisit();
    newVetVisit.setId(350);
    newVetVisit.setNotes("Talks a lot");
    newVetVisit.setSymptoms("Sore throat");

    newPet.getVetVisits().addElement(newVetVisit);
    newVetVisit.setPet(newPet);
    newPet.setPetOwner(newPetOwner);

    uow.registerNewObject(newPet);
uow.commit();
```

However, after the Unit of Work, the variables `newPet`, `newPetOwner`, and `newVetVisit` should not be used since they were technically copies from the Unit of Work. If we needed a handle to the cache version of these business objects we could query for them or we could have done the Unit of Work as follows:

```
UnitOfWork uow = session.acquireUnitOfWork();
    Pet newPet = new Pet();
    Pet newPetClone = (Pet)uow.registerObject(newPet);
    newPetClone.setId(150);
    newPetClone.setType("Horse");
    newPetClone.setName("Ed");

    PetOwner newPetOwner = new PetOwner();
    PetOwner newPetOwnerClone =
        (PetOwner)uow.registerObject(newPetOwner);
    newPetOwnerClone.setId(250);
    newPetOwnerClone.setName("George");
    newPetOwnerClone.setPhoneNumber("555-9999");

    VetVisit newVetVisit = new VetVisit();
    VetVisit newVetVisitClone =
        (VetVisit)uow.registerObject(newVetVisit);
    newVetVisitClone.setId(350);
    newVetVisitClone.setNotes("Talks a lot");
    newVetVisitClone.setSymptoms("Sore throat");

    newPetClone.getVetVisits().addElement(newVetVisitClone);
    newVetVisitClone.setPet(newPetClone);
    newPetClone.setPetOwner(newPetOwnerClone);
uow.commit();
```

Common Errors

Trying to use a new object after a Unit of Work if the new object was registered with `registerNewObject` or was not registered but persisted by being reachable. These objects should not be used after committing the Unit of Work.

ADVANCED API

Using registerAllObjects

This example examines how to use the advanced API method `registerAllObjects`.

The `registerAllObjects` method takes a Collection of objects as an argument and returns a Collection of clones. This allows you to register many objects at once. For example:

```
UnitOfWork uow = session.acquireUnitOfWork();
Collection toRegister = new Vector(2);
VetVisit vv1 = new VetVisit();
vv1.setId(70);
vv1.setNotes("May have flu");
vv1.setSymptoms("High temperature");
toRegister.add(vv1);

VetVisit vv2 = new VetVisit();
vv2.setId(71);
vv2.setNotes("May have flu");
vv2.setSymptoms("Sick to stomach");
toRegister.add(vv2);

uow.registerAllObjects(toRegister);
uow.commit();
```

ADVANCED API

Using Registration and Existence Checking

This example examines how to use an existence checking policy to accelerate object registration.

About Existence Checking

When TopLink writes an object to the database, TopLink runs an existence check to determine whether to perform an insert or an update. By default, TopLink uses the "check cache" existence checking policy. You can specify the default existence checking policy for a project as a whole or on a per-descriptor basis. The existence checking policy options are:

- check cache
- check database
- assume existence
- assume non-existence

If you use any existence checking policy other than "check cache", then you can use the way you register your objects to your advantage to reduce the time it takes TopLink to register an object.

Check Database

If your existence checking policy is "check database" then TopLink will check the database for existence for all objects registered in a Unit of Work. However, if you know that an object is new or existing, rather than use the basic `registerObject` method, you can use `registerNewObject` or `registerExistingObject` to bypass the existence check. TopLink will not check the database for existence on objects that

you have registered these methods. It will automatically do an insert if registerNewObject is called or an update if registerExistingObject is called.

Assume Existence

If your existence checking policy is “assume existence” then all objects registered in a Unit of Work are assumed to exist and TopLink will always do an update to the database on all registered objects, even new objects if you registered them with registerObject. However, if you use the registerNewObject method on the new object, TopLink knows to do an insert in the database even though the existence checking policy says “assume existence”.

Assume Non-existence

If your existence checking policy is “assume non-existence” then all objects registered in a Unit of Work are assumed to be new and TopLink will always do an insert to the database, even on objects read from the database. However, if you use the registerExistingObject method on existing objects, TopLink knows to do an update to the database.

ADVANCED API

Working with Aggregates

Aggregate mapped objects should never be registered in TopLink (in fact, you will get an exception if you try). Aggregate cloning and registration is automatic based on the owner of the aggregate object. In other words, if you register the owner of an aggregate, the aggregate is automatically cloned. When you get a working copy of an aggregate owner, its aggregate is also a working copy.

The bottom line of working with aggregates is you should always use an aggregate within the context of its owner:

- If you get an aggregate from a working copy owner, then the aggregate is a working copy.
- If you get an aggregate from a cache version owner then the aggregate is the cache version.

ADVANCED API

Unregistering Working Clones

This example examines how to unregister a clone from a Unit of Work.

The Unit of Work unregisterObject method allows you to unregister a previously registered object from a Unit of Work. An unregistered object will be ignored in the Unit of Work and any uncommitted changes made to the object up to that point will be discarded.

In general, this method is rarely used. It can be useful if you create a new object, but then decide to delete it in the same Unit of Work (which is also not recommended).

ADVANCED API

Declaring Read-Only Classes

This example examines how to use the advanced API methods `addReadOnlyClass` and `addReadOnlyClasses` to improve Unit of Work performance.

If you know that a class will not be modified, you can communicate this to the Unit of Work before you register an instance of that class. When a class has been identified to a Unit of Work as read only, then the Unit of Work will ignore:

- instances of that class
- instances of any class it refers to
- instances of any class that extends it

That means if you register instances of such objects (or if they are registered implicitly), the Unit of Work will not create or merge clones for these instances. They will not participate in the transaction and any changes you make to these objects will not be persisted.

For example, suppose class A owns a class B and class C extends class B. You know that only instances of A will be changed: you know that no class B's will be changed. Before registering an instance of class B, you can use:

```
myUnitOfWork.addReadOnlyClass(B.class);
```

Then you can proceed with your transaction: registering A objects, modifying their working copies, and committing the Unit of Work.

At commit time, the Unit of Work will not have to compare backup copy clones with the working copy clones for instances of class B (even if instances were registered explicitly or implicitly). This can improve Unit of Work performance if the object tree is very large.

Note that if you registered an instance of class C, the Unit of Work would not create or merge clones for this object and any changes you made to your C would not be persisted. This is because C extends B and B was identified as read-only.

If you have more than one class you want to identify as read only, you can add them to a Vector and use:

```
myUnitOfWork.addReadOnlyClasses(myVectorOfClasses);
```

If you have one or more classes you want to identify as read-only for every Unit of Work, use the Project method `setDefaultReadOnlyClasses(Vector)`.

ADVANCED API

Conforming Queries

This example examines how to include new objects in queries within a Unit of Work prior to commit, including:

- "Using Conforming Queries"
- "Conforming Query Alternative"

Using Conforming Queries

Because queries are executed on the database, querying through a Unit of Work will not, by default, include new, uncommitted, objects in the Unit of Work. The Unit of Work will not spend time executing your query against new, uncommitted, objects in the Unit of Work unless you explicitly tell it to.

Assume that a single Pet of type Cat already exists on the database. Examine the code shown in the following example.

```
UnitOfWork uow = session.acquireUnitOfWork();
Pet pet2 = new Pet();
Pet petClone = (Pet)uow.registerObject(pet2);
petClone.setId(200);
petClone.setType("Cat");
petClone.setName("Mouser");

ReadAllQuery readAllCats = new ReadAllQuery();
readAllCats.setReferenceClass(Pet.class);
ExpressionBuilder builder = new ExpressionBuilder();
Expression catExp = builder.get("type").equal("Cat");
readAllCats.setSelectionCriteria(catExp);

Vector allCats = (Vector)uow.executeQuery(readAllCats);

System.out.println("All 'Cats' read through UOW are: " + allCats);
uow.commit();
```

This produces the following output:

```
All 'Cats' read through UOW are: [Pet type Cat named Fluffy id:100]
```

If you tell the query "readAllCats" to include new objects:

```
readAllCats.conformResultsInUnitOfWork();
```

The output would be:

```
All 'Cats' read through UOW are: [Pet type Cat named Fluffy id:100, Pet
type Cat named Mouser id:200]
```

Bear in mind that conforming will impact performance. Before you use conforming, make sure that it is actually necessary. For example, consider the alternative described in "Conforming Query Alternative".

Conforming Query Alternative

Sometimes, you need to provide other code modules with access to new objects created in a Unit of Work. Conforming can be used to provide this access. However, the following alternative is significantly more efficient.

Somewhere a Unit of Work is acquired from a Session and is passed to multiple modules for portions of the requisite processing:

```
UnitOfWork uow = session.acquireUnitOfWork();
```

In the module that creates the new employee:

```
Pet newPet = new Pet();
```

```
Pet newPetClone = (Pet)uow.registerObject(newPet);
uow.setProperty("NEW PET", newPet);
```

In other modules where newPet needs to be accessed for further modification, it can simply be extracted from the Unit of Work's properties:

```
Pet newPet = (Pet) uow.getProperty("NEW PET");
newPet.setType("Dog");
```

Confirming queries are ideal if you are not sure if an object has been created yet or the criteria is dynamic.

However, for situations where the quantity of objects is finite and well known, this simple and more efficient solution is a very practical alternative.

ADVANCED API

Using commitAndResume

This example examines how to use the advanced API method commitAndResume.

A Unit of Work can be committed and allowed to resume. By calling commitAndResume instead of calling commit, the database and cache will be updated as normal but the Unit of Work is still considered open and further changes can be made. All the clones are still valid and you do not need to re-register objects in the resumed Unit of Work.

For example:

```
UnitOfWork uow = session.acquireUnitOfWork();
PetOwner petOwnerClone =
    (PetOwner)uow.readObject(PetOwner.class);
petOwnerClone.setName("Mrs. Newowner");
uow.commitAndResume();
petOwnerClone.setPhoneNumber("KL5-7721");
uow.commit();
```

The commitAndResume call produces the SQL:

```
UPDATE PETOWNER SET NAME = 'Mrs. Newowner' WHERE (ID = 400)
```

And then the commit call produces the SQL:

```
UPDATE PETOWNER SET PHN_NBR = 'KL5-7721' WHERE (ID = 400)
```

ADVANCED API

Using a Nested Unit of Work

This example examines how to nest one Unit of Work within another.

TopLink fully supports nested units of work. To create a nested Unit of Work you acquire a Unit of Work from another Unit of Work using the acquireUnitOfWork method.

Note that TopLink does not update the database or the cache until the outermost Unit of Work is committed.

Working copies from one Unit of Work are not valid in another, even between an inner and outer Unit of Work. You must register objects at all levels of a Unit of Work where they are is used.

For example:

```
UnitOfWork outerUOW = session.acquireUnitOfWork();
    Pet outerPetClone = (Pet)outerUOW.readObject(Pet.class);

    UnitOfWork innerUOWa = outerUOW.acquireUnitOfWork();
        Pet innerPetCloneA =
            (Pet)innerUOWa.registerObject(outerPetClone);
            innerPetCloneA.setName("Muffy");
        innerUOWa.commit();

    UnitOfWork innerUOWb = outerUOW.acquireUnitOfWork();
        Pet innerPetCloneB =
            (Pet)innerUOWb.registerObject(outerPetClone);
            innerPetCloneB.setName("Duffy");
        innerUOWb.commit();
    outerUOW.commit();
```

ADVANCED API

Using a Unit of Work with Custom SQL

This example examines how to emit custom SQL within the context of a Unit of Work.

You can add custom SQL to a Unit of Work at any time by calling the Unit of Work `executeNonSelectingCall` method as shown in the following example:

```
uow.executeNonSelectingCall(new SQLCall(mySqlString));
```

ADVANCED API

Using a Unit of Work with JTA

This example examines how to use a Unit of Work with Java Transaction API (JTA)/Java Transaction Service (JTS) with Container Managed Persistence (CMP).

When you configure a Unit of Work to use the JTA/JTS services of the host application server, you delegate transaction control and demarcation to the J2EE container. The Unit of Work participates in JTA/JTS transactions but does not control them.

In this case, you use the Session `getActiveUnitOfWork` method to acquire a Unit of Work associated with the container's active transaction. The container ensures that the JDBC connection provided is bound by the active transaction.

For two-phase commit, the J2EE container manages the transaction and the Unit of Work is just a participant. To use two-phase commit, you must register multiple resources into the same JTA transaction. When the container performs commit processing, it makes the appropriate call backs to both data sources and registered listeners — such as `TopLink`. When `TopLink` receives the appropriate call back, it issues its SQL against the database.

This section describes:

- "Configuring a Unit of Work to Use JTA in sessions.xml"

- "Configuring a Unit of Work to Use JTA in Java"
- "Acquiring a Unit of Work in a JTA Environment"
- "Using a Unit of Work with an Existing JTA Transaction"
- "Using a Unit of Work with a New JTA Transaction"

Configuring a Unit of Work to Use JTA in sessions.xml

To configure the use of an external transaction controller in the sessions.xml file:

1. Configure a JTA-enabled data source on your application server according to your J2EE container documentation (in this example, it is named "MyApplicationDS")
2. Configure the login element of your sessions.xml file to reference this data source, the use of an external transaction controller, and the use of an external connection pool:

```
<data-source>jdbc/MyApplicationDS</data-source>
<uses-external-transaction-controller>
  true
</uses-external-transaction-controller>
<uses-external-connection-pool>
  true
</uses-external-connection-pool>
```

3. Add an external transaction controller to your sessions.xml file:

```
<external-transaction-controller-class>
oracle.OraclerASTopLink.jts.oracle9i.Oracler9iJTSExternalTransactionController
</external-transaction-controller-class>
```

Configuring a Unit of Work to Use JTA in Java

To configure the use of an external transaction controller in the sessions.xml file:

1. Configure a JTA-enabled data source on your application server according to your J2EE container documentation (in this example, it is named "MyApplicationDS").
2. Configure the Login to specify a DataSource, the use of an external transaction controller, and the use of an external connection pool:

```
login.setConnector(
    new JNDIConnector(new InitialContext(), "jdbc/MyApplicationDS")
);
login.setUsesExternalTransactionController(true);
login.setUsesExternalConnectionPooling(true);
```

3. Configure the session to use a particular instance of ExternalTransactionController:

```
serverSession.setExternalTransactionController(
    new Oracler9iJTSExternalTransactionController()
);
```

Acquiring a Unit of Work in a JTA Environment

You use a Unit of Work to write to a database even in a JTA environment. To ensure that only one Unit of Work is associated with a given transaction, use the `getActiveUnitOfWork` method to acquire a Unit of Work as shown in the following example:

```
boolean shouldCommit = false;
// Read in any pet.
Pet pet = (Pet)clientSession.readObject(Pet.class);
UnitOfWork uow = clientSession.getActiveUnitOfWork();
// If uow is not null, use existing external transaction
if (uow == null) {
    // Start external transaction
    uow = clientSession.acquireUnitOfWork();
    shouldCommit = true;
}
Pet petClone = (Pet) uow.registerObject(pet);
petClone.setName("Furry");
if (shouldCommit) {
    uow.commit(); // Ask external transaction controller to commit
}
}
```

The `getActiveUnitOfWork` method searches for an existing external transaction:

- If there is an active external transaction and a Unit of Work is already associated with it, return this Unit of Work.
- If there is an active external transaction with no associated Unit of Work, then acquire a new Unit of Work, associate it with the transaction and return it.
- If there is no active external transaction in progress, return null.

If a non-null Unit of Work is returned, use it exactly as you would in a non-JTA environment: the only exception is that you do not call the commit method (see "Using a Unit of Work When an External Transaction Exists").

If a null Unit of Work is returned, start an external transaction either explicitly through the `UserTransaction` interface, or by acquiring a new Unit of Work using the `acquireUnitOfWork` method on the client session (see "Using a Unit of Work When No External Transaction Exists").

Using a Unit of Work When an External Transaction Exists

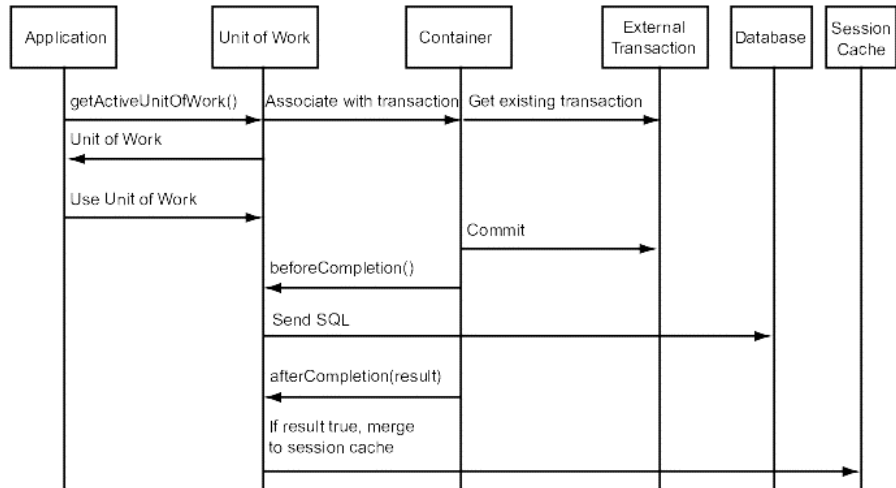
When `getActiveUnitOfWork` returns a non-null Unit of Work, you are associated with an existing external transaction. Use the Unit of Work as usual.

As the external transaction was not started by the Unit of Work, issuing a commit on it will not cause the JTA transaction to be committed. The Unit of Work will defer to the application or container that began the transaction. When the external transaction does get committed by the container, `TopLink` receives synchronization callbacks at key points during the commit.

The Unit of Work sends the required SQL to the database when it receives the `beforeCompletion` call back.

The Unit of Work uses the boolean argument received from the afterCompletion call back to determine if the commit was successful (true) or not (false).

If the commit was successful, the Unit of Work merges changes to the session cache. If the commit was unsuccessful, the Unit of Work discards the changes.



Using a Unit of Work When No External Transaction Exists

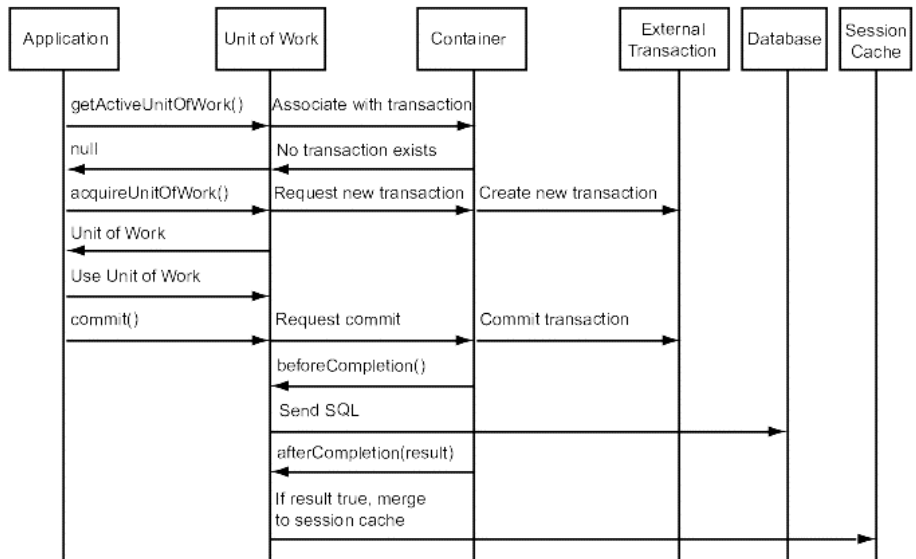
When getActiveUnitOfWork returns a null Unit of Work, there is no existing external transaction. You must start a new external transaction.

Do this either by starting an external transaction explicitly using the UserTransaction interface, or by acquiring a new Unit of Work using the acquireUnitOfWork method on the server session.

Use the Unit of Work as usual.

Once the modifications to registered objects are complete, you must commit the transaction either explicitly through the UserTransaction interface or by calling the Unit of Work commit method.

The transaction synchronization callbacks are then invoked on TopLink and the database updates and cache merge occurs based upon those callbacks.



ADVANCED API

Controlling the Order of Deletes

This example examines the advanced API methods you can use to control the order in which a Unit of Work deletes objects.

In "Deleting an Object", we learned that TopLink always properly orders the SQL based on the mappings and foreign keys in your object model and schema. You can control the order of deletes using:

- "Unit of Work setShouldPerformDeletesFirst Method"
- "Descriptor addConstraintDependencies Method"

Unit of Work setShouldPerformDeletesFirst Method

It is possible to tell the Unit of Work to issue deletes before inserts and updates by calling the Unit of Work setShouldPerformDeletesFirst method.

By default, TopLink does inserts and updates first to ensure that referential integrity is maintained.

If you are replacing an object with unique constraints by deleting it and inserting a replacement, if the insert occurs before the delete, you may raise a constraint violation. In this case, you may need to call setShouldPerformDeletesFirst so that the delete is performed before the insert.

Descriptor addConstraintDependencies Method

The constraints used by TopLink to determine delete order are inferred from one-to-one and one-to-many mappings. If you do not have such mappings, you can add constraint knowledge to TopLink using the descriptor addConstraintDependencies(Class) method.

For example, suppose you have a composition of objects: A contains B (one-to-many, privately owned) and B has a one-to-one, non-private relationship with C.

You want to delete A (and in doing so the included B's) but before deleting the B's, for some of them (not all) you want to delete the associated object C.

There are two possible solutions:

- "Using deleteAllObjects without addConstraintDependencies "
- "Using deleteAllObjects with addConstraintDependencies"

Using deleteAllObjects without addConstraintDependencies

In the first option, we do not use privately-owned on the one-to-many (A to B) relationship. When deleting an A, we make sure to delete all of it's B's as well as any C instances. For example:

```
uow.deleteObject(existingA);
uow.deleteAllObjects(existingA.getBs());
// delete one of the C's
uow.deleteObject(((B) existingA.getBs().get(1)).getC());
```

This option produces the following SQL:

```
DELETE FROM B WHERE (ID = 2)
DELETE FROM B WHERE (ID = 1)
DELETE FROM A WHERE (ID = 1)
DELETE FROM C WHERE (ID = 1)
```

Using deleteAllObjects with addConstraintDependencies

In the second option, we keep the one-to-many (A to B) relationship privately owned and add a constraint dependency from A to C. For example:

```
session.getDescriptor(A.class).addConstraintDependencies(C.class);
```

Now the delete code would be:

```
uow.deleteObject(existingA);
uow.deleteAllObjects(existingA.getBs());
// delete one of the C's
uow.deleteObject(((B) existingA.getBs().get(1)).getC());
```

This option produces the following SQL:

```
DELETE FROM B WHERE (A = 1)
DELETE FROM A WHERE (ID = 1)
DELETE FROM C WHERE (ID = 1)
```

As you can see, in both cases, the B is deleted before A and C. The main difference is that the second option will generate fewer SQL statements as it knows that it is deleting the entire set of B's related from A.

CONCLUSION

The TopLink Unit of Work is a flexible and powerful tool that simplifies the task of transactionally modifying objects directly or by way of a J2EE external transaction controller. In a TopLink application, it is the preferred method of writing to a database because it:

- sends a minimal amount of SQL to the database during the commit by updating only the exact changes down to the field level
- reduces database traffic by isolating transaction operations in their own memory space
- optimizes cache synchronization, in applications that use multiple caches, by passing change sets (rather than objects) between caches
- isolates object modifications in their own transaction space to allow parallel transactions on the same objects
- ensures referential integrity and minimizes deadlocks by automatically maintaining SQL ordering
- orders database inserts, updates, and deletes to maintain referential integrity for mapped objects
- resolves bi-directional references automatically

BEST PRACTICES

This section lists recommendations to bear in mind while using the Unit of Work.

1. Get into the pattern of always using clones (see "Creating an Object").
2. Although you can query through a Unit of Work and get back clones to save the registration step, it is wise to design your application to do the ReadAllQuery through a Session and then register only the objects that need to be changed (see "Modifying an Object").
3. Primary key based queries, by default, return what is in the cache but non-primary key based queries return what is in the database (see "Determining Whether or not an Object is the Cache Object").
4. Although TopLink exceptions are instances of RuntimeException and therefore do not need to be caught, you should handle exceptions thrown by the Unit of Work commit method (see "Handling Exceptions").
5. Typically you register a new object when you want to retain a handle to the cache version of the object after commit. However, when associating a new object that has been registered with a Unit of Work, bear in mind that TopLink treats the new object as if it was a clone: before commit, you have a reference to a cache object; after commit, that reference refers to a clone (see "Associations: New Target to Existing Source Object").
6. When associating a new target to an existing source object, even if you do not register the object, TopLink creates a clone for you (your cache object reference becomes a clone reference). However, when associating a new source object with an existing target object, you must register the new object (see "Associations: New Source to Existing Target Object").
7. Use conformResultsInUnit of Work to include uncommitted objects in queries executed within a Unit of Work but bear in mind this feature has a performance impact (see "Conforming Queries").
8. Use commitAndResume instead of commit if you need to continue changing after a commit: all currently registered clones remain valid and do not need to be re-registered (see "Using commitAndResume").
9. If you must pass a clone into an object's constructor and then register the new object, use registerNewObject (see "Using registerNewObject").
10. Never register an Aggregate mapped object. Bear in mind that if you get an aggregate from a working copy owner then the aggregate is a working copy; if you get an aggregate from a cache version then the aggregate is the cache version (see "Working with Aggregates").
11. In a nested Unit of Work, TopLink does not commit until the outermost Unit of Work is committed. You must register an object at each level of Unit of Work where it is used (see "Using a Nested Unit of Work").

GLOSSARY

Term	Definition	Also Known As ...
Cache Version	The original business object stored at the root session level in the TopLink cache. This is the business object of which TopLink preserves identity throughout the JVM. Often referred to simply as "Cache Copy".	Master version, original version, identity version
Existing Object	An object that is already persistent in the database. This would include a new object after it has been successfully written to the database or an object read from the database.	Persistent object, previously written object
Demarcate	To determine the beginning and end of a transaction. In a JTA/JTS transaction, the J2EE container is responsible for demarcating a transaction. Otherwise, you use the Unit of Work commit (or commitAndResume) method to define the end of a transaction.	
Source	The originator object of a relationship. For example, if an Employee has a 1-1 relationship to an Address, we say that the Employee is the source. If a Teacher has a 1-M relationship with Student, we say the Teacher is the source.	
Target	The opposite of the source object of a relationship. For example, if an Employee has a 1-1 relationship to an Address, we say that the Address is the target. If a Teacher has a 1-M relationship with Student, we say the Student is the target.	
Unit of Work	A transaction at the object level following typical transaction ACID properties (Atomicity, Concurrency, Isolation and Durability).	"Object level" transaction, TopLink transaction, UOW, transactional context
Unit of Work Working Copy	A working copy of your business object that can be modified in isolation in a Unit of Work. Typically referred to simply as a "Clone".	Clone, working copy, copy



TopLink Unit of Work Primer
August 2003
Author: Peter Purich
Contributing Authors: Tatjana Obradovic, Douglas Clarke

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Copyright © 2003, Oracle. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.