Java Versant Interface Usage Manual

Release 7.0.1.4

VERSANT

Versant History, Innovating for Excellence

In 1988, Versant's visionaries began building solutions based on a highly scalable and distributed object-oriented architecture and a patented caching algorithm that proved to be prescient.

Versant's initial flagship product, the Versant Object Database Management System (ODBMS), was viewed by the industry as the one truly enterprise-scalable object database.

Leading telecommunications, financial services, defense and transportation companies have all depended on Versant to solve some of the most complex data management applications in the world. Applications such as fraud detection, risk analysis, simulation, yield management and real-time data collection and analysis have benefited from Versant's unique object-oriented architecture.

For more Information please visit **www.versant.com**

Versant US

Versant Corporation
255 Shoreline Drive, Suite 450, Redwood City, CA 94065
Ph +1 650-232-2400, Fx +1 650-232-2401

Versant Europe

Versant GmbH
Wiesenkamp 22b, 22359 Hamburg, Germany
Ph +49.40.60990-0, Fx +49.40.60990-113

© 2008 Versant Corporation.
All products are trademarks or registered trademarks of their respective companies in the United States and other countries.

The information contained in this document is a summary only.

For more information about Versant Corporation and its products and services, please contact Versant Worldwide or European Headquarters.

Table of Contents

CHAPTER 1

Overview of JVI

This Chapter gives detailed explanation of the basic concepts and architecture of JVI.

The Chapter covers the following in detail:

- Introduction
- Architecture
- Contents of the JVI Usage Manual

INTRODUCTION

The Java Versant Interface (JVI) combines the best features of the Java programming language and the Versant object database system to provide efficient, easy-to-use storage of persistent objects.

JVI is built on Java, a language whose flexibility and performance has proven valuable on all levels of complex 2- and 3- tier applications. JVI embraces the Java philosophy, providing persistence that fits naturally in the Java language:

- All JVI programs are written using pure Java syntax and semantics. No special keywords have been added to the language and no awkward interface is needed to access persistent data.
- Objects of nearly any Java class can be stored and accessed persistently in the database. Most kinds of data can be stored, including elemental values such as strings and integers and Java references to other objects. With JVI, you can access the fields of a persistent object directly in the application without writing tedious data mapping routines.
- Java garbage collection allows you to focus on solving problems instead of low-level memory management issues. JVI exposes persistent objects directly as Java objects and works seamlessly with the garbage collector.
- Java has built-in support for multiple threads of execution working simultaneously in an application. JVI allows each thread to operate in shared or independent transactions.

JVI is tightly integrated with the Versant object database, a state-of-the-art database system with a comprehensive set of features:

- High-performance, fully transactional data storage, access and query with multiple programming language interfaces.
- An advanced 2-tier architecture with full caching of objects on both client and server. All cache synchronization occurs automatically at transaction boundaries.
- Sophisticated locking models including standard 2-phase and optimistic locking.
- Event notifications (such as creation, deletion and modification) to database clients.
- Fault-tolerant data replication with fast online failover operation.
- A rich object model including class inheritance and direct object references.

ARCHITECTURE

Bindings

The JVI functionality is split into two main levels:

- Fundamental and
- Transparent

The Fundamental binding underlies the entire system, providing a core foundation of database access routines.

The Transparent binding builds upon Fundamental, mapping persistent objects directly into the Java object space. Together they make database storage in Java simple, flexible and efficient.

JVI also supports the industry-standard ODMG Java interface. By adhering to the ODMG standard, you can use the Versant database without becoming “locked in” to a proprietary interface. The ODMG binding is implemented as a thin layer on top of the Transparent binding.

In addition, JVI provides several tools and utility classes. These include collection classes such as list, set, vector and table, application development utilities and class libraries for special functionality such as event notification. These utilities work with both Fundamental and Transparent layers to extend the power of JVI.

Balanced Client-Server Network Architecture

JVI is an application interface to the Versant object database, whose structure consists of multiple database servers connecting to remote application clients. The Versant client library caches objects, providing fast object access and navigation within the application, while database queries can be executed at the server using the VQL query language. Unlike a typical relational database system, the server will not ship a database object to the client if the object is already contained in the object cache.

Database Access with the Fundamental and Transparent Bindings

The JVI Fundamental binding exposes persistent objects through a powerful programmatic interface. Java objects encapsulate the various types of objects in the database. For example, Versant sessions have corresponding `Session` objects, persistent schema and instance objects in the database are referenced using `ClassHandle` and `Handle` objects, individual fields are read and modified with `Attr` objects, and queries are performed with query and `Predicate` objects.

The power and flexibility of the Fundamental binding allows applications to access the Versant database in a systematic manner, without compiled-in knowledge of the schema for each persistent object stored in the database. This makes the Fundamental binding useful for tools, utilities and middleware applications.

The Fundamental binding is not ideally suited for typical application development, because it leaves a significant amount of work to the application programmer. In many respects, most database interfaces suffer from these same problems; for example, you must use get and set methods to access persistent attributes and need to laboriously write “mapping” code that translates the data between persistent and in-memory formats.

The Transparent binding helps to solve this problem by merging the Java and database object spaces. With the Transparent binding, persistent objects are seen in the application as regular Java objects. JVI automatically manages the reading, writing and locking of persistent objects as they are accessed by the application.

JVI allows an application to combine the Fundamental and Transparent models within a single application or transaction. Therefore you can easily choose between the power and flexibility of the Fundamental binding and the ease-of-use and performance of the Transparent binding.

CONTENTS OF THE JVI USAGE MANUAL

This User Guide describes the essential concepts of the Java Versant Interface, and gives a task-oriented “how to” for many of the concepts. The “how to” provides a simple example that can be used as a basic model when learning about JVI or writing a JVI application. For information on each Java class and method in the JVI class library, see the JVI javadoc.

The User Guide is split into following chapters:

- Chapter 1: **Overview of JVI**, which you are reading now. It provides a high-level overview of the basic concepts and architecture of JVI.
- Chapter 2: **Fundamental JVI,** describes the JVI Fundamental binding in detail. Since the Fundamental binding is an underlying component of the entire JVI system, this chapter contains a large amount of useful information. On the other hand, it is not necessary to completely understand the Fundamental binding to use the Transparent binding. If you plan to use the Transparent binding, you can skip this chapter and begin with Chapter 3.
- Chapter 3: **Transparent JVI,** explains how to use the JVI Transparent binding. This chapter is of particular importance for developers who are planning to use the Transparent binding.
- Chapter 4: **ODMG JVI,** describes the ODMG Standard Interface that is available with JVI.
- Chapter 5: **Utilities**, covers the utility classes that are part of the JVI interface, including interfaces for database administration and collection of statistical information.
- Chapter 6: **Event JVI,** describes the JVI Event notification.
- Chapter 7: **Java Connector Architecture (JCA),** describes the JVI JCA connectors.
- Chapter 8: **Versant JTA/XA,** describes JTA/XA support in JVI.
- Chapter 9: **Session Managed Persistence,** describes the SMP.

CHAPTER 2

Fundamental JVI

This Chapter gives us detailed explanation about the concepts of “Fundamental JVI”.

The Chapter covers the following in detail:

- Introduction
- Basic Concepts and Tasks
- Queries 7.0
- Queries 6.0
- Advanced Concepts

INTRODUCTION

Overview

The Fundamental JVI enables you to directly access Versant ODBMS functionality.

To understand how Fundamental JVI can provide direct access to Versant ODBMS, you should know the:

- JVI view of Versant ODBMS
- General design of class hierarchy in Fundamental JVI

Fundamental JVI view of Versant ODBMS

Fundamental JVI is a semi-direct wrapper around the Versant C API and provides access to most Versant functionality.

Fundamental JVI exposes aspects of the Versant object model and interface (sessions, attributes and queries) as Java objects.

Persistent objects in the database are not seen as Java objects. You can access the attributes of a persistent object by invoking the `get()` and `put()` methods.

Fundamental JVI supports only Versant-supported data types.

General design of class hierarchy in Fundamental JVI

In Fundamental JVI, most public types are Java interfaces (purely abstract types). You can create private concrete implementations of the public interfaces with factory methods in the interfaces.

What's in this chapter

This chapter contains explanations of specific basic and advanced concepts and shows you how to use Fundamental JVI based on the concepts.

Basic Concepts and Tasks

Concepts	**Tasks**
Sessions and Transactions	**Using a database session** • How to instantiate and start a session • How to instantiate and start a session with a list of properties • How to commit changes to the database with standard operations • How to roll back a transaction • How to use a savepoint • How to end a database session
Handles and Objects	**Using `Handle` for persistent object operations** • How to use `Handle` to get LOID information • How to create a `Handle` with LOID information • How to use `Handle` to get schema information from an instance • How to use `Handle` to read from and write to instances • How to use `Handle` to delete instances • How to use `Handle` to upgrade or downgrade a lock • How to use `Handle` to check front-end object caching • How to determine if a `Handle` is empty
Class Handles and Class Objects	**Using `ClassHandle` for database operations** • How to use `Builders` to define schema • How to get class objects from the database • How to use `ClassHandle` to get class information • How to use `ClassHandle` to create and delete an index • How to use `ClassHandle` to create instances • How to use `ClassHandle` to delete a class and all instances
`Attrs` and Attributes	**Using `Attr` objects** • How to create `Attr` objects

Queries	**Querying the database** • How to select class instances • How to enumerate query results • How to construct a predicate • How to use predicates for path queries • How to use a cursor to select objects • How to use additional select options • How to use VQL for a query
Locking	**Using locking** • How to set a default lock mode • How to set and release locks on a persistent object
Error Handling	**Handling errors** • How to use `VException` to handle exceptions

Advanced Concepts and Tasks

Concepts	**Tasks**
Threads and Multiple Sessions	**Using threads in a session** • How to attach threads to a session • How to detach threads from a session • How to use one thread per session • How to share Fundamental JVI classes across threads
Multiple Databases	**Using multiple databases** • How to connect with multiple databases • How to disconnect multiple databases • How to get schema from another database • How to synchronize classes
Memory Management	**Using memory management APIs** • How to get object cache usage

Advanced Transactional Methods	**Explanation only**
Security	**Setting security for a session** • How to create a secure session
Optimistic Locking	**Using optimistic locking** • How to use optimistic locking
Schema Evolution	**Modifying class definitions** • How to use ClassHandle to modify class definitions
Object Migration	**Explanation only**
Group Operations	**Using `HandleVector` to group handles** • How to use `HandleVector` to access elements • Using `HandleVector` to perform group operation • How to read a group of objects • How to write a group of objects • How to delete a group of objects • How to migrate a group of objects • How to copy a group of objects • How to update a group of objects in the object cache • How to release a group of objects from the object cache • How to zap a group of objects from the object cache

BASIC CONCEPTS AND TASKS

Sessions and Transactions

Overview

A session connects Fundamental JVI to Versant ODBMS and administers all database activities.

You can connect to Versant ODBMS in a single Versant session or in multiple concurrent sessions.

To explain sessions and transactions, this section describes:

- The relationship between sessions and transactions
- The object cache and persistent objects
- Session classes and the session interface

To explain how to use a database session, this section also describes:

- How to instantiate and start a session
- How to instantiate and start a session with a list of properties
- How to commit changes to the database with standard operations
- How to roll back a transaction
- How to use a savepoint
- How to end a database session

Relationship between sessions and transactions

When you begin a session, Versant tracks your activities in a transaction. As a result, a one-to-one relationship develops between a session and a transaction. Each session corresponds to a transaction and each transaction associates with a session.

For more information, about sessions and transactions, please refer to the *Versant Database Fundamentals Manual*.

The following describes what happens to a transaction during a session:

If you . . .	**then you . . .**
begin a session	begin a transaction.
end a session	commit a transaction.
commit (roll back) a transaction	begin a new transaction automatically.

The object cache and persistent objects

Each session maintains an object cache of persistent objects. The C (native) heap supports this cache as in the case of when you use Versant C APIs.

The object cache of persistent objects has the following characteristics:

- Occupies memory and contributes to the process size.
- Consumes none of the Java heap.

The Versant application process parameters control the cache size.

For more information, on cache size, please refer to Chapter "Database Profiles" in the *Versant Database Administration Manual*.

How to instantiate and start a session

The `Session` interface encapsulates transactional behavior. The `FundSession` class implements the `Session` interface.

Whether your application uses a single Versant session, multiple concurrent sessions or multiple threads that share one session, you create a database session by instantiating an instance of `FundSession`. You can then use methods in `FundSession` to manage your sessions and threads.

The following example uses the constructor that takes a database name as an argument:

```
FundSession session = new FundSession ("mydb@server");
```

How to instantiate and start a session with a list of properties

You can start a session by initializing specific properties associated with the session opened in the Versant database, such as database name, session name and session options.

An instance of `java.util.Properties` can optionally be provided as an argument to a `Session` constructor. The properties are set as key-value pairs.

For example:

```
Properties prop = new java.util.Properties ();
// the name of the database
prop.put ("database", "mydb@server");
// a read-only database session
prop.put ("options", Constants.READ_ACCESS + "");
// no implicit locking
prop.put ("lockmode", Constants.NOLOCK + "");
FundSession s = new FundSession (p);
// If option Constants.RELEASE_SCHEMA_OBJ is set, the schema objects
are released
prop.put ("options", Constants.RELEASE_SCHEMA_OBJ + " ");
FundSession s = new FundSession (p);
```

Options are defined in the `Constants` class. Since you must supply a `string`, you can convert the `Constants` integer values to a `String` by appending `"+"`. To specify multiple session options, use the "|" symbol to concatenate them.

For a complete list of Session properties and options, please refer to the description of "`FundSession`" in the *JVI javadoc.*

NOTE:-

Since the constants for session option and lock mode are integers, you need to convert them to `string`.

Session names must be unique in the same Java Virtual Machine. If not provided, a default session name is assigned.

How to commit changes to the database with standard operations

You must open a transaction in your application to access, create or modify a persistent object. A thread must explicitly create a transaction by instantiating a session or associate itself with an existing transaction by joining a session.

To commit changes to the database with standard operations, you can use the following methods:

```
commit()
checkpointCommit()
```

`commit()`

You can use the `commit()` method to:

- Update the database with all persistent modifications made in the transaction.
- Delete objects marked for deletion.
- Release any locks held by the transaction.
- Drop all persistent objects from the object cache.
- If `Constants.RELEASE_SCHEMA_OBJ` is set, while creating a session, then release the schema objects, else retain the schema objects in the objects cache.
- Start a new transaction.

`checkpointCommit()`

You can use the `checkpointCommit()` method to:

- Update the database with all persistent modifications made in the transaction.
- Delete objects marked for deletion.
- Maintain short locks.
- Maintain the object cache.
- Start a new transaction.

How to roll back a transaction

You can use the `rollback()` method to:

- Discard database actions made since the last commit.
- Release short locks.

- Drop all persistent objects from the object cache.
- Unmark the objects marked for deletion.
- Start a new transaction.

How to use a savepoint

You can use the `savepoint()` method to :

- Create a snapshot of database conditions.
- Return to those database conditions without ending the transaction.

The `undoSavepoint()` method returns database conditions to those preceding the immediately previous savepoint. If no immediately previous savepoint exists, conditions are restored to those that existed at the last `commit()`.

How to end a database session

You can use the `endSession()` method to:

- Commit the current transaction.
- Clear the COD table in the object cache.
- Terminate the session and close communication channels to the database.

You should not use the session object after this method invocation. In addition, you should not use any objects that reference this session, such as `Handles.`

Handles and Objects

Overview

A `Handle` represents a link or reference to a persistent object stored in the Versant ODBMS and processes all access to persistent objects.

You can associate each `Handle` with only one session.

To explain `Handles`, this section describes:

- The relationship between `Handles` and `Sessions`
- The functionality of `Handles`
- The identity of `Handles`

To explain how to use `Handle` for persistent object operations, this section describes:

- How to use `Handle` to get LOID information
- How to create a `Handle` with LOID information
- How to use `Handle` to get schema information from an instance
- How to use `Handle` to read from and write to instances
- How to use `Handle` to delete instances
- How to use `Handle` to upgrade or downgrade a lock
- How to use `Handle` to check front-end object caching
- How to determine if a `Handle` is empty

Relationship between Handles and Sessions

A `Handle` object contains a reference to a database session where the `Handle` object is created. A `Handle` is only valid when you use it with that session. In addition, you must not use it any more after its session ends.

Functionality of Handles

Handles are designed to:

- Access attributes of a persistent object with `get()` and `put()` methods.
- Explicitly set and release locks on a persistent object.
- Control the caching of a persistent object.

Identity of Handles

You can identify `Handles` by the LOID of a persistent object to which it references.

The `asByteArray()`, `asString()` and `asLong()` methods can obtain a LOID (in different forms) from a `Handle` object.

The `newHandle()` methods on `Session` can create a `Handle` from a LOID.

A Handle can be "empty". An empty Handle indicates that it references the null Versant object, with LOID 0.0.0.

For Handles, the standard Java methods equals() and hashCode() are defined with the LOID of the persistent object to which the Handle refers. According to the equals() method, two Handles are "equal" if they reference the same persistent object and associate with the same session.

How to get LOID information

You can use Handle to obtain LOID information on a persistent object.

For example:

```
// LOID as a String
String loidAsString = handle.asString ();
// LOID as a long value
long loidAsLong = handle.asLong ();
// LOID as a byte array
byte[] loidAsBytes = handle.asByteArray ();
```

How to create a Handle

You can create a Handle object from a LOID represented as a string, long, or byte array.

For example:

```
HandlehandleFromString = session.newHandle (loidAsString);
HandlehandleFromLong   = session.newHandle (loidAsLong);
HandlehandleFrombBytes = session.newHandle (loidAsBytes);
```

How to get schema information from an instance

You can use Handle to obtain information related to the class of a persistent object.

For example:

```
// To obtain the class name of this persistent object
String className = handle.classNameOf ();
// To obtain the class object of this persistent object
ClassHandle cls = handle.classObjectOf ();
```

You can use the following method to determine whether a given Handle refers to a class object.

For example:

```
boolean b = handle.isClassObject ();
```

How to read and write instances

You need to create Attr helper objects for each attribute in the class before you access the attributes in a persistent instance of the class.

See also "Attrs and Attributes" on page 44.

Suppose you have a Student class defined as follows:

```
class Student {
String[]  subjects;
   String    description;
   float     gpa;
   Student[] allStudents;
}
```

If you have a handle to a Student object in the database, you can use the get() and put() methods on the Handle to retrieve and store its attributes.

For example:

```
AttrStringList  subs = session.newAttrStringList("subjects");
AttrString      desc = session.newAttrString      ("description");
AttrFloat       gpa  = session.newAttrFloat       ("gpa");
AttrHandleArray all  = session.newAttrHandleArray ("allStudents");
handle.put (desc, "has a lean and hungry look");
float gotGradePoint = handle.get (gpa);
handle.put (gpa, 3.56);
Handle[] v = handle.get (all);
String[] str = handle.get(subs); //  It will return all the subjects
```

The above example sets the description attribute of the Student instance to the string "has a lean and hungry look", obtains the gpa attribute, sets it to the value 3.56, and retrieves the allStudents attribute in an Handle array.

Notice that when the get() and put() methods are invoked on the Handle to the instance, the Attr helper, which corresponds to the database attribute name and type, is the argument to both methods. The exact type of the `Attr` object determines the type of value stored by the `put()` method (the value is the second argument) and type of the value returned by the `get()` method.

In general to insert a Java type into a persistent instance attribute, you can use:

```
handle.put (anAttrInstance, value);
```

To retrieve a persistent instance attribute as a Java type, you can use:

```
value = handle.get (anAttrInstance);
```

NOTE:- The `put()` method attempts to acquire a write lock on the database object. The changes made by the `put()` method are not permanent until the database transaction is committed.

How to delete instances

You can invoke the `deleteObject()` method on a `Handle` to delete a persistent object.

For example:

```
handle.deleteObject ();
```

NOTE:-

- If the server profile parameter `commit_delete` is `OFF`, this function will send the delete request to the source database and the object is deleted immediately.
- If `commit_delete` is `ON`, this function will acquire a Write Lock on the object and set its status as "`Marked for deletion`". The object will be physically deleted at commit.

  If the transaction commits, the objects are physically deleted. If a rollback occurs, these objects are un-marked and their status restored.
- Queries run on the database will not include objects marked for deletion in the result sets.

How to upgrade or downgrade a lock

You can invoke the following methods to upgrade or downgrade the lock on a persistent object:

For example:

```
// downgrade the lock so there is no longer a lock on the object
handle.downgradeLockTo (Constants.NOLOCK);
// upgrade the lock to a write lock and return immediately
handle.upgradeLockTo (Constants.WLOCK, Constants.LK_NOWAIT);
```

NOTE:- To obtain a lock on an object which was not previously locked, the method `refreshObject(int lockmode)` may be a better choice than the `upgradeLockTo()` method.

If an object is not locked, another transaction could change it; thus, creating a stale copy. If you use the `refreshObject()` method, it provides the latest copy of the object.

How to check front-end object caching

You can check to see if a persistent object has been cached in the front-end object cache.

For example:

```
boolean isCached = handle.isCached ();
```

You can check to see if a persistent object has been pinned in the front-end object cache.

For example:

```
boolean isPinned = handle.isPinned ();
```

You can find and pin an object in the front-end object cache with a given lock mode.

For example:

```
// The object is brought to the front-end cache,
// pinned, with the write lock mode set on it
handle.locateObject (Constants.WLOCK);
```

You can update a persistent object in the front-end object cache with the current state in the database; pin the object in the cache; and set a given lock mode on the object.

For example:

```
// The object state is refreshed and a write lock set on it
handle.refreshObject (Constants.WLOCK);
```

You can use the `releaseObject()` method on the `Handle` to release an object from the front-end object cache. This method releases a previously pinned or dirtied persistent object. The `releaseObject()` method is also typically used when there is no further use for an object in a transaction. Since the object could be dirty, you must write out this object if you want to write the changes to the database before you release it. You must use a group-write operation to write out the object.

For example:

```
// The object is released from the C front-end cache
handle.releaseObject ();
```

You can use the method `int FundSession.getCodCount (int category)` to get the count of cached objects descriptors of a given category that are in the cached object descriptor table.

Valid input categories of the CODs are :

1. Constants.COD_ITER_ALL: To get count of All cached object descriptors.
2. Constants.COD_ITER_CACHED: To get count of Cached object descriptors of objects currently in the object cache.
3. Constants.COD_ITER_DELETED: To get count of Cached object descriptors for objects marked for deletion
4. Constants.COD_ITER_DIRTY: To get count of Cached object descriptors for objects marked dirty

For more details refer the javadoc for FundSession.

How to determine if a Handle is empty

A `Handle` is empty when it references the `null` Versant object with LOID 0.0.0.

A `Handle` to a Versant `null` LOID `(0.0.0)` is the constant empty handle.

For example:

```
Handle handle = session.emptyHandle();
```

The `isEmpty()` method checks whether a `Handle` is empty or not.

NOTE:- An empty `Handle` is not a `null` Java object.

How to redefine equals() and hashCode()

To define the standard Java methods `equals()` and `hashCode()`, you can use the LOID of the persistent object to which the `Handle` refers . According to the `equals()` method, two `Handles` are equal if they reference the same persistent object and associate with the same session.

Class Handles and Class Objects

Overview

A `ClassHandle` represents a link to a Versant schema, class or object.

The `ClassHandle` interface extends the `Handle` interface. All operations on `Handle` apply to `ClassHandle`.

Each `ClassHandle` is associated with a single database.

Different `ClassHandles` can exist for the same class name when the classes exist in different databases.

All operations on the `ClassHandle` occur according to the associated database.

You can use `ClassHandles` to perform the following tasks:

- Finding superclasses and subclasses of the class
- Creating new instances of the class
- Performing queries on the class
- Creating indexes on attributes of the class

To explain how to use `ClassHandle` for database operations, this section describes:

- How to use `Builders` to define schema
- How to use `ClassHandle` to get schema from the database
- How to use `ClassHandle` to get class information
- How to use `ClassHandle` to create and delete an index
- How to use `ClassHandle` to create instances
- How to use `ClassHandle` to delete a class
- How to use `ClassHandle` to delete all instances

How to use Builders to define schema

The Builder interfaces encapsulate the functionality of building classes and attributes by providing an intermediate type.

The ClassBuilder interface provides methods that help define a class, such as specifying attributes, superclasses and the method to define a class itself. The Session interface extends the ClassBuilder interface. Therefore, you can invoke methods provided by ClassBuilder on an instantiated session.

The following Session methods return a ClassBuilder :

```
withAttrBuilders()
withSuperBuilders()
defineClass()
```

The AttrBuilder interface serves as an intermediate type when defining an index on an attribute. The category of newAttr* methods on a Session such as: newAttrInt, newAttrFloat and so on, return an AttrBuilder instance. You can use an AttrBuilder as an intermediary type.

For example, suppose you have the following Java class:

```
class Student extends Person {
String[] teachers;
   String   description;
   float    GPA;
   Course[] allCourses;
}
```

You can use an array of SuperBuilder to name the superclass.

```
SuperBuilder supers[] = { session.newSuperBuilder("Person") };
```

You can use an array of AttrBuilder to name and type the attributes.

```
AttrStringList teach = session.newAttrStringList     ("teachers");
AttrString      desc    = session.newAttrString      ("description");
AttrFloat       gpa     = session.newAttrFloat       ("gpa");
AttrHandleArray courses = session.newAttrHandleArray ("AllCourses");

AttrBuilder[] attrs = {
```

```
session.newAttrBuilder(teach),
   session.newAttrBuilder (desc),
   session.newAttrBuilder (gpa).withBTreeIndex (),
   session.newAttrBuilder (courses)
};
```

The `with()` methods can modify `AttrBuilder` if you want indexes:

```
.withBTreeIndex()
.withHashIndex()
```

You can use the `SuperBuilder` and `AttrBuilder` arrays to define the class:

```
ClassHandle student = session.withSuperBuilders (supers)

withAttrBuilders (attrs)
.defineClass ("Student");
```

The `defineClass()` method takes the name of the class being defined, and returns a `ClassHandle` to the new class.

How to get class objects

To find a class that has already been defined to the database, you can use the `locateClass()` method on `Session`.

The `locateClass()` method locates and returns a `ClassHandle` to the class named in the argument from the session database.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
```

How to use ClassHandle to get class information

The `ClassHandle` interface exposes methods to obtain class related information:

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
// To obtain the class name
String className = cls.classname ();
// To obtain the database name
```

```
String dbName = cls.database ();
```

The `ClassHandle` interface contains a `superclasses()` method that returns a `ClassHandle` array of superclasses. Similarly, the `subclasses()` method returns an array of `ClassHandles` to subclasses. The array can be traversed to access individual `ClassHandles` to the superclasses or subclasses respectively.

Since Fundamental JVI presents you with the view of the Versant database, a persistent class could have multiple immediate superclasses, although the Java language does not support multiple inheritance.

For example, if you access a class defined from C++ that has multiple inheritance, the `superclasses()` method would return multiple `ClassHandles`.

```
// To obtain the sub-classes
ClassHandle[] subCls = cls.subclasses ();
// To obtain the super-classes
ClassHandle[] superCls = cls.superclasses ();
```

How to use ClassHandle to create and delete index

Indexes can improve query performance. However, indexes are not necessary to perform a query on a database class.

An attribute can have up to two indices:

- B-tree index (either normal or unique)
- Hash table index (either normal or unique).

You can create a new index on an attribute in a class.

For example:

```
ClassHandle cls = session.locateClass ("Person");
// Defining a Unique B-Tree Index on the "name attribute"
cls.createIndex ("name", Constants.UNIQUE_BTREE_INDEX);
```

NOTE:- Indexes do not have names.

You can delete the index on an attribute in a class.

For example:

```
// Deleting the index on the "name" attribute
cls.deleteIndex ("name", Constants.UNIQUE_BTREE_INDEX);
```

How to use ClassHandle to create instances

You need an instance of a `ClassHandle` to make an instance.

To find out how you can obtain a `ClassHandle,` see "*How to use Builders to define schema"* and *"How get class objects from the database"* in this chapter.

You can make an instance by invoking `makeObject()` on a `ClassHandle` to obtain a `Handle` to the new instance.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
Handle obj = cls.makeObject ();
```

The new instance is created in the same database as the source database of its `ClassHandle`. The newly created database instance does not have its attribute values set yet.

For an explanation of how you can set attribute values of a persistent object, please refer to section "How to read and write instances" on page 35.

How to use ClassHandle to delete a class and all instances

You can invoke the `dropClass()` method on the `ClassHandle` to delete a class with all its instances.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
cls.dropClass ();
```

You can invoke the `dropInstances()` method on the `ClassHandle` to delete all the instances of a class in a database. Unlike `dropClass()`, `dropInstances()` leaves the class object in the database.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
cls.dropInstances ();
```

Attrs and Attributes

Overview

An Attr object references an attribute of a database class. In Versant, database attributes are strongly typed, thus each Attr helper is strongly typed to correspond to a database attribute type.

Attr objects are initialized with the name of the attribute they reference. They do not know any specific class or database. However, because they can cache information, Attr objects perform best if they are used on instances of the same class.

The Attr helper instance manages access to attributes in a persistent object. Classes prefixed with "Attr" such as AttrInt, AttrFloat and so on, are Attr helper classes.

You can use Attr objects to:

- Specify what attributes a class has when creating a database class.
- Specify which attributes to access when reading or writing the attribute value of an instance.
- Specify which attributes to query when querying a class in the database.

Mapping

Each Attr object is strongly typed to map with a corresponding Versant database attribute type. The following table shows the mapping:

Java Type	JVI Fundamental Type	Versant Database Type
Boolean or boolean	AttrBoolean	o_bool
Boolean[] or boolean[]	AttrBooleanArray	o_bool[]
Boolean[] or boolean[]	AttrBooleanList	o_bool[list]
Byte or byte	AttrByte	o_1b
Byte[] or byte[]	AttrByteArray	o_1b[]
Byte[] or byte[]	AttrByteList	o_1b[list]
Double or double	AttrDouble	o_double
Double[] or double[]	AttrDoubleArray	o_double[]
Double[] or double[]	AttrDoubleList	o_double[list]

Float or float	AttrFloat	o_float
Float[] or float[]	AttrFloatArray	o_float[]
Float[] or float[]	AttrFloatList	o_float[list]
Integer or int	AttrInt	o_4b
Integer or int[]	AttrIntArray	o_4b[]
Integer[] or int[]	AttrIntList	o_4b[list]
java.sql.Date	AttrInt	o_date
java.sql.Date[]	AttrIntArray	o_date[]
java.sql.Date[]	AttrLongList	o_date[list]
java.sql.Time	AttrInt	o_time
java.sql.Time[]	AttrIntArray	o_time[]
java.sql.Time[] AttrIntList o_time[list]	AttrIntList o_time[list]	o_time[list]
java.sql.TimeStamp	AttrLong	o_timestamp
java.sql.TimeStamp[]	AttrLongArray	o_timestamp[]
java.sql.TimeStamp[]	AttrLongList	o_timestamp[list]
java.util.Date	AttrLong	o_timestamp
java.util.Date[]	AttrLongArray	o_timestamp[]
java.util.Date[]	AttrLongList	o_timestamp[list]
Long or long	AttrLong	o_8b
Long[] or long[]	AttrLongArray	o_8b[]
Long[] or long[]	AttrLongList	o_8b[list]
Object Reference (Link)	AtrrHandle	o_object
Object Reference (Link) Array	AttrHandleArray	o_object[]
Object Reference (Link) Array	AttrHandleList	o_object[list]
Short or short	AttrShort	o_2b
Short[] or short[]	AttrShortArray	o_2b[]
Short[] or short[]	AttrShortList	o_2b[list]

String	AttrString	(UTF-8 encoded) char []
String	AttrString8Bit	char []
String[]	AttrStringList	char[list]

NOTE:- Java has no unsigned types corresponding to the Versant `o_u1b`, `o_u2b` and `o_u4b` types. This binding accesses the above Versant types as signed Java `byte`, `short` and `int` respectively. Anomalous results are expected when the high bit is set. The methods like `add()`, `remove()`, `size()`, `indexOf()` etc. can be used on List typed datatypes.

Using Attr object

The following describes how to create `Attr` objects.

How to create Attr objects

The `Session` interface defines factory methods to create `Attr` objects. These methods are named `newAttrType`, where `Type` is the attribute. For example, `newAttrInt` creates and returns an instance of `AttrInt`.

For example, suppose you have a `Student` class defined in your database as follows:

```
class Student {
String[]  subjects;
   String    description;
   float     gpa;
   Student[] allStudents;
}
```

You can then create `Attr` helpers as follows:

```
AttrStringList subs = session.newAttrStringList   ("subjects");
AttrString      desc = session.newAttrString      ("description");
AttrFloat       gpa  = session.newAttrFloat       ("gpa");
AttrHandleArray all  = session.newAttrHandleArray ("allStudents");
```

QUERIES 7.0

Introduction

The basic goal for Queries 7.0 is to significantly increase the expressive power of Versant's Query engine, as well as to improve the performance.

To harness the power of the new C APIs designed for query processing, we have introduced following new classes in fundamental layer of JVI i.e. in `com.versant.fund package`.

- **FundQuery**

  It encapsulates one query represented by a query string.
- **FundQueryResult**

  This instance is the result of a query execution. It is an encapsulation of the result set and serves as an iterator.
- **QueryExecutionOptions**

  This constructor returns a new instance of `QueryExecutionOptions` with all options set to false. The options are encapsulated this way to make sure that only correct options can be passed and no bit operations need to be used to enable several options.

NOTE:- Queries 7.0 is the newer version of queries which users are encouraged to use as Queries 6.0 might be deprecated in future.

FundQuery

Constructor

```
FundQuery(TransSession session, java.lang.String queryString)
FundQuery(TransSession session, java.lang.String database,
 java.lang.String queryString)
```

The constructor without the database parameter uses the default database of the session. The constructor will compile the query internally and throw an exception if the compilation fails.

Getters

`java.lang.String getDatabase()`

Returns the name of the database in which the query will run.

`java.lang.String getQueryString()`

Returns the query string.

`FundSession getSession()`

Returns the session.

Sole getter methods do not have corresponding setter methods because the query is compiled inside the constructor.

Setter / Getter

`int getClassLockMode()`

Returns the class lock mode used by this query.

`FundQuery setClassLockMode(int mode)`

Sets the class lock mode for this query.

`int getInstanceLockMode()`

Returns the instance lock mode used by this query.

`FundQuery setInstanceLockMode(int mode)`

Sets the instance lock mode for this query.

The lock to be placed on returned instances.

By definition, a select operation returns reference to the objects that satisfy your predicate. The default is to not place a lock on the selected objects themselves. If you specify a read, update, or write lock in this parameter, objects are locked by the server if they satisfy the predicate.

This means that during the time of the select operation, you may encounter lock conflicts. It also means that afterwards, you are guaranteed that the objects corresponding to your returned result are consistent and will not change while you are looking at them.

By comparison, if you do not specify a lock in this parameter and then use the returned references to fetch objects in a separate operation, there will be a window of time in which other users can modify or delete your selected objects. Even if you specify fetching options or a non-zero value in levels, a window of time will still exist in which your selected objects may be modified (although it will be much smaller.).

For NOLOCK the fetching and calculation of the closure will use the default lock mode of the session, which is typically a RLOCK. If you want to perform dirty reads then use the session method `setDefaultLock`(NOLOCK) before executing the query.

`int getFetchSize()`

Gets the fetch size currently used by this query.

`FundQuery setFetchSize(int size)`

Sets the fetch size of this query.

`QueryExecutionOptions getQueryExecutionOptions()`

Gets the execution options used by this query.

`FundQuery setQueryExecutionOptions(QueryExecutionOptions options)`

Sets the execution options for this query.

`FundQuery setCandidateCollection(java.lang.String candidate, Handle[] handles)`

Sets the candidate collection for this query.

`Handle[] getCandidateCollection(java.lang.String candidate)`

Gets the Candidate collection used by this query.

Methods

```
FundQueryResult execute()
FundQueryResult execute(QueryExecutionOptions options)
```

The `FundQuery.execute()` method will use the default `QueryExecutionOptions` as provided by the static method `QueryExecutionOptions.defaultOptions()`.

```
void bind(String parameter, Object value)
```

`FundQuery.bind()` can take any single value parameter. With this method, all Date and Time classes from JVI will be translated directly into the correct type needed for the query. Hence users now need not call any methods from class DateTimeConvert.

`FundQuery.bind()` will return the Query instance to allow chaining of setter and bind methods.

Clean up

```
void close()
```

Closes the result set and frees internal resources.

NOTE:- Closing a query is important to prevent memory leaks.

FundQueryResult

Methods

boolean isEmpty()

Determines if this FundQueryResult is empty

Handle next()

Returns the next handle in the result set, or null if no handles are available to return anymore.

Handle[] next(int size)

Returns up to size handles from the result set into an array

Handle[] nextAll()

Returns the remaining handles in the result set.

FundQuery getQuery()

Returns the `FundQuery` that created this `FundQueryResult`.

void close()

Closes the result set and frees internal resources.

`FundQueryResult` encapsulates the query result and serves as an iterator through the result set. It allows the retrieval of individual handle with the `next()` method as well as batch operation with the next (int size) method. The `next(int size)` method will return up to size handles. The `nextAll()` method will return all remaining handles in the `FundQueryResult`.

public Object nextRow()

Returns the next row in the result set, or **null** if no handles are available to return anymore.

```
return The next row or null.
```

where the return value of nextRow() is:

- type Handle in case of no projections,
- wrapper type in case of projection on base type fields,
- Handle in case of reference field
- Object [] in case of more than one column, where each element of the Object [] is of the type as appropriate to the respective column type.

public Object[] nextRows(int size)

Returns up to *size* rows from the result set into an array.

If there are less rows than *size* left in the result set, the array will contain the remaining rows left in the QueryResult.

If there were no rows left in the query result, an empty array is returned.

param size - The maximum number of rows returned in the array.

```
return An array of Object[] instances of the result set.
```

If there is only one column selected than the object of appropriate type is returned for each row.

The length of the array is at most *size*.

public Object[] allRows()

Returns the remaining rows in the result set.

If no calls to nextRow(), nextRows(size) or allRows() have been made before, this method will return all rows in the original result set.

If there were no rows left in the query result, an empty array is returned.

```
return An array of Object instances of the result set.
```

QueryExecutionOptions

Constructor

```
QueryExecutionOptions()
```

This constructor returns the default instance of the class `QueryExecutionOptions`.

Setter / Getter

```
boolean isFetchObjects()
QueryExecutionOptions setFetchObjects(boolean fetchObjects)
```

If set, the query will load all objects into the client cache immediately. The fetch operation will place the instanceLockmode on the objects. If the instanceLockmode is NOLOCK then the default lock mode of the session is used.

```
boolean isFlushAll()
QueryExecutionOptions setFlushAll(boolean flushAll)
```

If set, the query will flush all new and dirty objects from the client cache first before executing the query in the server.

```
boolean isFlushNone()
QueryExecutionOptions setFlushNone(boolean flushNone)
```

If set, the query will not flush new and dirty objects from the client cache before executing the query in the server.

```
boolean isPinObjects()
QueryExecutionOptions setPinObjects(boolean pinObjects)
```

If set, objects loaded with fetch all are also pinned in the C Cache

NOTE:- `FlushAll` and `FlushNone` are mutually exclusive. Setting one of the options to true will automatically set the other option to false if set.

Usage Examples

Simple query

```
FundQuery query = new FundQuery(session_, "select selfoid from
 test.model.Order where date_ > $date");

query.bind("date", new java.sql.Date(105, 3, 10)); // April 10th 2005

FundQueryResult result = query.execute(new

QueryExecutionOptions().setFetchObjects(true));
Handle hnd;
int count = 0;
while ((hnd = result.next()) != null) {
 count++;
}
System.out.println("Found  : " + count);
query.close();
```

Batch Query

```
FundQuery query = new FundQuery(session_, "select selfoid from
 test.model.Order where date_ > $date");

query.bind("date", new java.sql.Date(105, 3, 10)); // April 10th 2005

FundQueryResult result = query.execute(new
queryExecutionOptions().setFetchObjects(true));

int batchSize = 50;

while (true) {
Handle[] hnd = result.next (batchSize);
displayBatch(objs);
if (hnd.length < batchSize) {
break;
}
}
result.close();
query.close();
```

Collection Query

```
FundQuery query = new FundQuery(session_, "select selfoid from
 test.model.Order where date_ > $date");

query.bind("date", new java.sql.Date(105, 3, 10)); // April 10th 2005

FundQueryResult result = query.execute();

Handle[] hnd = result.nextAll();

processOrders(Arrays.asList(obj));

query.close();
```

QUERIES 6.0

Overview

Queries allow you to find persistent objects that match a specified condition.

Queries are evaluated in the Versant back-end (server). The query results are passed to the front-end (client).

To explain queries, this section describes:

- Predicate objects
- Comparison Operators
- Logical Operators
- VQL
- HandleEnumeration
- Indexes

To explain how to query the database, this section describes:

- How to select class instances
- How to enumerate query results
- How to construct a predicate
- How to use a predicate for path queries
- How to use a cursor to select objects
- How to use additional select options
- How to use VQL for a query
- How to use cursors with VQL queries

Predicate objects

A predicate object encapsulates a selection condition or criteria used to examine object values.

A predicate object can be simple or complex. The following describes simple and complex predicates:

- You can create simple predicate objects by calling Comparison Operator methods such as, `eq` (equals), `ne` (not equals) and `ge` (greater than or equal) on the `Attr` object for the attribute to which the predicate should be applied.
- You can form complex predicate objects by joining together simpler predicates with Logical Operator methods on predicate.

To find the objects of a class whose attributes satisfy your predicate, pass the predicate object to the `select()` method of the `ClassHandle` for that class.

Comparison Operators

JVI supports the following comparison operator methods in the family of `Attr` interfaces to determine how an attribute value is compared to a key value:

Relational operators:	`eq()`
	`ne()`
	`lt()`
	`le()`
	`gt()`
	`ge()`
String pattern operators:	`matches()`
	`notMatches()`
Class membership operators:	`isa()`
	`notIsa()`
Set operators:	`intersects()`
	`notIntersects()`
	`supersetOf()`
	`notSupersetOf()`

```
SubsetOf()
NotSubsetOf()
EquivalentSetOf()
NotEquivalentSetOf()
```

Logical operators

Multiple, simple predicates can be combined with the following logical operators:

```
and()
or()
not()
```

See also “How to construct a predicate” on page 60.

VQL Queries

VQL (Versant Query Language) specifies search criteria as strings instead of predicate objects.

Fundamental JVI expresses VQL queries by creating a `FundVQLQuery` object. The constructor accepts a query string as argument.

Some values, such as arrays and strings, are difficult or impossible to express in VQL. To solve this problem, use *substitution parameters*. A substitution parameter is written as `"$1", "$2"`, and so on, within the query string. You can then substitute the values with the `bind()` method on the `FundVQLQuery` object.

For example:

```
FundVQLQuery vql = FundVQLQuery (session,
   "select selfoid from Person where name = $1 and age > $2");
vql.bind ("John");
vql.bind (21);
HandleEnumeration e = vql.execute ();
while ( e.hasmoreHandles() ) {
   Handle handle = e.nexthandle();
   System.out.println (handle);
}
```

NOTE:- Set operators, such as `intersects()` and `supersetOf()`, are only supported in predicate objects, but not in VQL.

HandleEnumeration

A `HandleEnumeration` allows you to iterate through all objects that result from a query.

Both methods, `ClassHandle.select()` and `FundVQLQuery.execute()`, return a `HandleEnumeration` object.

The `HandleEnumeration` interface extends the `java.util.Enumeration` interface. You can use it the same way you use an `Enumeration` object, except `HandleEnumeration` includes the `hasMoreHandles()` and `nextHandle()`methods.

`HandleEnumeration` also contains additional operations you can use to determine the size of the query result or access the entire result as an array.

Indexes

You can use an index on an attribute to make a query execute more quickly. However, using an index can increase overhead when you update objects.

An index can be designated as unique or not unique.

You can use the `createIndex()` and `deleteIndex()` methods on the `ClassHandle` to create or delete indexes.

Versant supports two types of indexes: B-tree and Hash indexes.

Index Type	Description
B-tree index	When a B-tree index exists, each object has an entry on a leaf page of a B-tree. The leaf pages are sorted on the B-tree attribute and doubly linked. A B-tree index is useful for value range comparisons and direct comparison of values.
Hash index	A Hash index is useful for direct comparison of values.

See also “How to use ClassHandle to create and delete index ” on page 42.

For more information on indexes and queries, please refer to the Chapter “Query Processing with VQL 7.0” in the *Versant Database Fundamentals Manual*.

How to select class instances

Selection of all instances of a class

You can invoke the `select()`method on the `ClassHandle` with no argument to select all instances of a class in a database. In the following example, the first statement finds the `Employee` class in the session database. The second statement returns the `Handles` of all `Employee` objects in the session database as an enumeration.

For example:

```
ClassHandle cls = session.locateClass ("Employee");
HandleEnumeration e = cls.select ();
```

Predicate based Selection

You can use predicate objects as predicates to form a query. These predicate objects are immutable. Predicates can be passed as an argument to a select statement. The following statement returns `Handles` of `Employee` objects in the session database that satisfy the predicate.

See also "How to construct a predicate" on page 60.

For example:

```
ClassHandle cls = session.locateClass ("Employee");
HandleEnumeration he = cls.select (predicate);
```

How to enumerate query results

You can enumerate query results by:

- Iterating through a `HandleEnumeration` one `Handle` at a time.
- Converting the `HandleEnumeration` to an array of `Handles` and accessing the array elements.
- Creating a `Vector` from the `HandleEnumeration` and iterating through the `Vector`.

Iterating through a `HandleEnumeration` one Handle at a time

You can iterate through one `Handle` at a time directly from the enumeration. The following example first finds all instances of a class named `"MyClass"`. It then enumerates through each `Handle` in the query result.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
HandleEnumeration e = cls.select ();
while ( e.hasMoreHandles() ) {
   Handle handle = e.nextHandle ();
   System.out.println (handle);
}
```

Converting enumeration to an array and access elements

You can convert the enumeration to an array and then access its elements.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
HandleEnumeration e = cls.select ();
Handle[] array = e.asArray();

for (int i = 0; i < array.length; i++) {
   System.out.println (array [i]);
}
```

Creating a Vector from enumeration and iterating through a Vector

You can create a `HandleVector` from the enumeration and iterate through the `HandleVector`.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
HandleEnumeration e = cls.select ();
HandleVector vector = session.newHandleVector (e);

for (int i = 0; i < hv.size(); i++) {
   Handle handle = vector.handleAt (i);
   System.out.println (handle);
}
```

How to construct a predicate

You can use predicates to specify selection criteria for objects. Invoking any of the following comparison methods on an `Attr` object creates a predicate. For a list of Comparison Operator methods, see "Predicate Objects" in this chapter. Each of the above comparison methods takes a value as an argument. The operator and the type of `Attr` determine the argument type.

For example:

A predicate for an integer attribute age where age is greater than 17:

```
Predicate p = session.newAttrInt("age").gt(17);
```

A predicate for a string attribute name that matches the regular expression `[Ss]tr*`:

```
Predicate q = session.newAttrString("name").matches("[Ss]tr*");
```

A predicate for a link attribute parent that is equal to the persistent object referred to by the `Handle` h:

```
Predicate r = session.newAttrHandle("parent").eq(h);
```

Predicates can be combined with the logical operator methods `or()`, `and()`, and `not()`.

For example:

```
Predicate x = p.and(q);
Predicate y = r.not();
Predicate z = x.or(y);
ClassHandle cls = session.locateClass ("MyClass");
HandleEnumeration e = cls.select (z);
```

How to use predicates for path queries

The `pathToType()` methods are defined on the `AttrHandle` and `AttrHandleArray` classes. These methods allow you to navigate through object reference attributes (links).

For more information about path queries, please refer to the *Versant Database Fundamentals Manual*.

For example:

```
class Employee {
   Garage[] garages;
   ...
}
class Garage {
   car[] cars;
   ...
}
class Car {
   Stringcolor;
   ...
}
```

Suppose you want to find all employees who own a red car according to the three classes defined above. Formulate a predicate as follows:

```
// Attr for the attribute Employee.garages
AttrHandleArray garages = session.newAttrHandleArray ("garages");
// Attr for the attribute Garage.cars
AttrHandleArray cars = session.newAttrHandleArray ("cars");
// Attr for the attribute Car.color
AttrString color = session.newAttrString ("color");
// Attr for the path from an Employee object to a Car object
AttrString garageToCarToColor =
   garages.pathToHandleArray(cars).pathToString(color);
// Find all employees with a red car in the garage
Predicate pred = garageToCarToColor.eq ("Red");
ClassHandle cls = session.locateClass ("Employee");
cls.select (pred);
```

How to use a cursor to select objects

With a large number of objects in a query, use a cursor-based query select method. A cursor select returns a `HandleEnumeration` just as a regular select, but it creates a cursor and no objects are fetched from the database until the next batch of objects is retrieved from the `HandleEnumeration`.

By fetching batches of objects at a time, rather than all the objects at once, a cursor select reduces lock contention on the fetched objects and lessens memory usage. When you finish using a cursor, you can invalidate it to release it.

For more information on cursor query, please refer to Section "Cursor Queries" in Chapter “Query Processing with VQL 6.0” in the *Versant Database Fundamentals Manual*.

The following example uses cursor-based query to find all instances of `Person`.

```
ClassHandle cls = session.locateClass ("Person");
```

```
HandleEnumeration people = cls.cursorSelect ();

while ( people.hasMoreHandles() ) {
   HandleVector hv = people.nextBatch();
   HandleEnumeration e = hv.handles();
   while ( e.hasMoreHandles() ) {
      Handle handle = e.nextHandle();
      System.out.println (handle);
   }
}
people.invalidateCursor ();
```

The following example shows how you can perform a cursor-based select with a predicate. Also, note that you can specify a batch size when retrieving the next batch from a `HandleEnumeration` obtained from a cursor select. The default batch size is 100.

```
ClassHandle cls = session.locateClass ("Person");
Predicate pred = session.newAttrInt("age").gt(18);
HandleEnumeration majors = cls.cursorSelect (pred);

while ( majors.hasMoreHandles() ) {
   HandleVector hv = majors.nextBatch (200);
   HandleEnumeration e = hv.handles ();
   while ( e.hasMoreHandles() ) {
      Handle handle = e.nextHandle ();
      System.out.println (handle);
   }
}
majors.invalidateCursor ();
```

NOTE:- Calls to API's `nextElement` and `nextBatch` should not co-exist on the same cursor object to fetch all instances satisfied by the cursor query. If used it can result in unpredictable behavior. Only use either `nextElement` or `nextBatch` in conjunction with / without `hashMoreElements`.

The following example uses cursor-based query to find all instances of Person.

```
ClassHandle cls = session.locateClass ("Person");
HandleEnumeration people = cls.cursorSelect ();

while (people.hasMoreElements ( ) ) {
  nextElement ( );
}
```

Or,

```
while (people.hasMoreElements ( ) ) {
  nextBatch  (10);
}}
```

How to use additional select options

You can use several optional methods before a select on a `ClassHandle`. For instance, the following example:

- Selects all instances of the `Person` class, in the database named `otherdb@myServer`.
- Sets an Intention-Read lock on the `Person` class.
- Sets a Read lock on all the selected instances.
- Uses a `DEEP_SELECT` option to return instances of all the subclasses of `Person`.
- Returns the `Person` instances and all objects that can be reached from them by setting select level to be -1.

```
ClassHandle cls = session.locateClass ("Person");
HandleEnumeration he = cls.withDatabase ("otherDb@myServer")
   .withSelectOptions (Constants.SELECT_DEEP)
   .withThisClassLockmode (Constants.IRLOCK)
   .withInstanceLockmode (Constants.RLOCK)
   .withSelectLevels (-1)
   .select ();
```

Select options

Options for the `select()` method are specified by the `withSelectOptions()`method. Possible select options are as follows:

Options	Description
`TOPIN`	Pin all the fetched objects in memory.
`SELECT_DEEP`	Evaluate and return subclasses.
`FLUSH_NONE`	Do not flush object cache before query.
`FLUSH_ALL`	Flush object cache before query.

These options can be combined with the "or" operator in Java, `"|"`.

For example:

```
.withSelectOptions (Constants.SELECT_DEEP |
                    Constants.FLUSH_ALL |
                    Constants.TOPIN)
```

Select levels

You can use the `.withSelectLevels()` method to specify how to traverse the number of link levels that start from objects found by the select predicate. You can use a 0 to return only the found objects and a -1 to return all objects linked directly or indirectly to the found objects. A positive number indicates the number of link levels to follow. These link levels start with the found objects.

The closure calculation will place the instanceLockmode on the objects. If the instanceLockmode is NOLOCK then the default lock mode of the session is used.

How to use VQL for a Query

The `FundVQLQuery` class allows you to create a query based on the Versant Query Language (VQL).

In general, you can use the following steps to use VQL:

- Construct a `FundVQLQuery` instance using a VQL select statement string.
- If using substitution parameters in the query string, you can use the `bind()` method to assign values. (The `ith` variable set in the query string is set by the `ith` call to the `bind()` method.)
- Execute the query to return an enumeration of query results.

For example:

```
FundVQLQuery vql = FundVQLQuery (session,
"select selfoid from Person where name = $1 and age $2");
vql.bind ("John");
vql.bind (21);
HandleEnumeration e = vql.execute ();
while ( e.hasMoreHandles() ) {
   Handle handle = e.nextHandle ();
   System.out.println (handle);
}
```

The general syntax of a query string is as follows:

```
select selfoid from [only] class [predicate]
```

For `class`, you must specify the name of the class on which the query operates. You can use the optional parameter `only` to exclude subclasses in the query.

For more information, on predicate VQL syntax, please refer "Versant Query Language 6.0" in the *Versant Database Fundamentals Manual*.

How to use Cursors with VQL Queries

To support cursor queries in FundVQLQuery two new methods have been added:

```
public HandleEnumeration executeWithCursor ( int batchSize ) ;
public HandleEnumeration executeWithCursor ( int options, int 
classLockMode, int instLockMode, int batchSize )
```

The following example uses cursor-based query in VQL:

```
FundVQLQuery vql = new FundVQLQuery (session, "select * from 
Person");
HandleEnumeration people = vql.executeWithCursor(100);
while ( people.hasMoreHandles() ) {
  HandleVector hv = people.nextBatch();
  HandleEnumeration e = hv.handles();
  while ( e.hasMoreHandles() ) {
    Handle handle = e.nextHandle();
    System.out.println (handle);
  }
}
people.invalidateCursor ();
```

For more information on cursor query, please refer to Section "Cursor Queries" in Chapter “Query Processing with VQL 6.0” in the *Versant Database Fundamentals Manual*.

Queries using Virtual Attribute Template (VAT)

Overview

To enhance query capability, the concept of Virtual Attributes has been introduced. A Virtual Attribute is a fictitious attribute that is not saved in the database but is computed on demand. Virtual Attributes can be indexed. These indices are maintained during normal object operations (insert, delete and update) and are used during query processing.

For more information, about Virtual Attributes and Virtual Attribute Templates (VAT), please refer to Section "Advanced Queries" in the Chapter "Query Processing with VQL 6.0" in the *Versant Database Fundamentals Manual.*

How to use Virtual Attributes in Predicate Terms

The attribute specified in the predterm can either be a virtual attribute or a real attribute.

See also "How to construct a predicate" on page 60.

An example of a predicate for a String attribute 'name', that uses the national VAT is as follows:

```
Predicate pred = session.newAttrString( "/national de_DE utf8 name"
 ).matches( "Wiederver*");
```

Using the nocase VAT we can query the database in a case insensitive mode. An example of a predicate for a `String` attribute `name` that uses the `nocase` VAT is as follows:

```
Predicate pred = session.newAttrString( "/nocase ascii name"
 ).matches("mortimer");
```

The tuple VAT can be used to specify multi attribute queries. An example of a predicate for a String attribute name that uses the tuple VAT is as follows:

```
Predicate pred  = session.newAttrString( "{/tuple firstName lastName"
 ).matches( "Carole Smith" );
```

Indices can be created on attributes that are either virtual or real. The code snippet below shows how an index can be created and deleted on a virtual attribute:

...

```
ClassHandlepersonHandle = session.withAttrBuilders(attrs).
defineClass("Person");
String vattr = “/national de_DE utf8 firstname”
personHandle.createIndex(vattr, Constants.UNIQUE_BTREE_INDEX);
…
personHandle.deleteIndex(vattr, Constants.UNIQUE_BTREE_INDEX);
```

How to use Virtual Attributes in VQL Queries

The example in the VQL query section above can be modified to use VAT as follows:

```
FundVQLQuery vql = FundVQLQuery (session,
"select selfoid from Person where ‘/national de_DE utf8 name‘ = $1 and
 age $2");
// name contains spaces, so use braces to indicate a single parameter.
vql.bind (“{Jürgen Uwe}”);
vql.bind (21);
HandleEnumeration e = vql.execute ();
```

NOTE:- Virtual attributes are enclosed in grave accents.

Locking

Overview

Locks provide guarantees that allow multiple processes to access the same objects in the same database at the same time in a co-operative, controlled and predictable manner. There are two types of locking models: optimistic and pessimistic.

This section discusses pessimistic locking.

See also “Optimistic locking” on page 90.

For general information, on optimistic and pessimistic locking, please refer to the Chapter “Locks” in the *Versant Database Fundamentals Manual*.

Pessimistic locking is the default locking model in Fundamental JVI.

To explain pessimistic locking, this section describes:

- Default lock
- Implicit locking
- Explicit locking
- Automatic dropping of read locks

To explain how to use locking, this section describes:

- How to set a default lock mode
- How to set and release locks on a persistent object

Default lock

Default lock is the lock (typically read lock) placed on objects that are fetched from the database. For example, when you access an object returned from a query, the object is brought in memory and a default lock is set on the object.

See also “How to set a default lock mode” on page 72.

Implicit locking

Implicit locking occurs when methods automatically obtain locks as needed.

For example:

- The transaction obtains the default lock on the object that contains the attribute when an attribute is read using the `Handle.get()` method.
- The transaction obtains a write lock on the object when an attribute is modified using the `Handle.put()` method.

The following Fundamental JVI methods obtain a lock implicitly:

Method	**Lock**
`Handle.deleteObject()`	write lock
`Handle.get()`	default lock
`Handle.markModified()`	write lock*
`Handle.put()`	write lock

`HandleVector.checkTimestamps()`	write lock
`HandleVector.groupDeleteObjects()`	write lock
`HandleVector.groupReadObjects()`	default lock
`FundSession.checkAllTimestamps()`	write lock

* In an optimistic locking session, the `WLOCK` is not acquired until commit.

The following Fundamental JVI methods release locks implicitly:

```
HandleVector.commitAndRetain()
HandleVector.rollbackAndRetain()
FundSession.commit()
FundSession.commitAndCleanCod()
FundSession.commitAndRetainSchema()
FundSession.commitAndRetain()
FundSession.endSession()
FundSession.rollback()
FundSession.rollbackAndRetain()
```

Explicit locking

Explicit locking occurs when methods specifically upgrade or downgrade locks.

Explicit locking affects the precedence assigned to a lock. For example, you can give higher precedence to a lock on an object (example: from no lock to write lock) with the `upgradeLockTo()` method on the `Handle`. You can also give lower precedence to a lock on an object (example: from read lock to no lock) with the `downgradeLockTo()` method on the `Handle`.

The following Fundamental JVI method sets a lock explicitly:

```
Handle.upgradeLockTo()
```

Constants defining all the lock modes are in the `Constants` interface. These are:

Lock	**Description**
`NOLOCK`	No lock
`WLOCK`	Write lock
`ULOCK`	Update lock
`RLOCK`	Read lock

`IWLOCK`	Intention write lock
`IRLOCK`	Intention read lock
`RIWLOCK`	Read intention write lock

The following Fundamental JVI methods release locks explicitly:

```
Handle.downgradeLockTo()
HandleVector.downgradeLocksTo()
```

The following Fundamental JVI methods allow you to specify a lock in a parameter:

```
ClassHandle.select()
Handle.locateObject()
Handle.refreshObject()
HandleVector.getClosure()
HandleVector.groupReadObjects()
HandleVector.refreshObjects()
```

The following schema evolution methods in the Fundamental JVI acquire a write lock on class objects:

```
ClassHandle.appendAttr()
ClassHandle.dropAttr()
ClassHandle.insertAttrsAt()
ClassHandle.renameAttr()
ClassHandle.renameClass()
```

Automatic dropping of read locks

After you invoke `FundSession.setThreadOptions(DROP_RLOCK)` in a thread, for the current session in this thread, the Versant server drops all read locks (`RLOCK` and `IRLOCK`) after instance objects are fetched from the database. You can use this option in either a standard session or an optimistic session.

Dropping read lock methods in Fundamental JVI

You can use the following methods to get, set and unset thread options:

```
FundSession.getThreadOptions()
FundSession.setThreadOptions()
FundSession.unsetThreadOptions()
```

The FundSession.setThreadOptions(DROP_RLOCK) method affects the following Fundamental JVI methods:

```
ClassHandle.select()
Handle.get()
Handle.locateObject()
Handle.refreshObject()
HandleEnumeration.nextBatch()
HandleVector.getClosure()
HandleVector.groupReadObjects()
HandleVector.refreshObjects()
```

How to set a default lock mode

You can specify the default lock mode of a session when creating the session. By default, a session is started with an RLOCK mode. The following example uses write lock as the default lock. Therefore, all objects accessed in this session, by default, have write locks.

For example:

```
Properties prop = new Properties ();
prop.put ("database", "mydb@myserver");
prop.put ("lockmode", Constants.WLOCK + "");
FundSession session = new FundSession (prop);
```

You can also get and set the default lock with method defaultLockMode() on the Session interface.

```
int currentLockMode = session.defaultLockMode ();
if (currentLockMode != Constants.RLOCK)
session.defaultLockMode(Constants.RLOCK);
```

How to set and release locks on a persistent object

You can use the upgradeLockTo() and downgradeLockTo() methods to set and release locks on a persistent object.

The upgradeLockTo() method explicitly sets a lock on the object referenced by the Handle. The downgradeLockTo() method downgrades to a weaker lock on the object.

Error Handling

Overview

Methods in Fundamental JVI do not return error codes. If an error occurs, a `VException` is thrown.

The message `string` for a `VException` displays the error code (an integer and a symbolic name) and a short description of the error meaning.

When catching a `VException`, the error code can be obtained with the `getErrno()` method.

Some database operations, such as `groupDeleteObjects()` in `HandleVector`, use a Vector of `Handles` in which objects that failed the operation are inserted. This Vector can be obtained with the `getFailedHandles()` method on returned `VException`.

How to use VException to handle exceptions

The `VException` class is a subclass of `java.lang.RuntimeException`. Therefore, your program need not have to explicitly enclose method invocations that could result in an exception within a `try/catch` block. All Versant kernel errors are thrown as `VExceptions`, and the corresponding kernel error number coincides with the `VException` error number. You can obtain this error number with `getError()`, and retrieve a string description with the `toString()`method.

For example:

```
try {
   ClassHandle cls = session.locateClass ("Apple");
   cls.dropClass();
} catch (VException exception) {
   // 6002 is the kernel error: "Class undefined"
   if (exception.getError() != 6002)
      throw exception;
   System.out.println ("Ignoring " + vex);
}
```

NOTE:- JVI could also throw Java exceptions such as, `OutOfMemoryError` or `NullPointerException` when appropriate.

Some database operations, in case of failure, also return a Vector containing objects that failed the operation. This Vector can be obtained by the `getFailedHandles()` method. This method gives a list of all failed objects, irrespective of the error.

Example for all Failed Objects:

```
Object[] objects = vex.getFailedObjects();
int count = 0;
int collectedCount = 0, collectedCount1=0;
 for (int i = 0; i < objects.length; i++)
{
      if( objects[i]!=null)
      {
        count++;
   }
  }
System.out.println("Failed objects are " + count);
```

Two other methods have been introduced to identify the objects failed with different type of errors produced, one for objects failed with error `OM_TR_INVALID_TS` and one for deleted objects.

Examples for Failed Objects:

objects = vex.getTimestampFailedObjects();

```
 if (objects!=null)
  {
   for (int i = 0; i < objects.length; i++)
   {
      collectedCount ++;
    }
  }
 System.out.println("TimeStamp Failed objects are " +
collectedCount);
```

Examples for Not Found Objects:

```
 objects = vex.getNotFoundObjects();
 if (objects!=null)
  {
      for (int i = 0; i < objects.length; i++)
       {
         collectedCount1 ++;
       }}
```

ADVANCED CONCEPTS

Threads and Multiple Sessions

Overview

In a multi-threaded application, a thread can be in only one session at a time, but multiple threads can share a single session. If the application uses multiple concurrent sessions, a thread can switch from one session to another.

To explain the relationship between threads and sessions, this section describes:

- Multiple Threads, Multiple Sessions (MTMS)
- Multiple Threads, Single Session (MT1S)

To explain how to use threads on a session, this section describes:

- How to attach threads to a session
- How to detach threads from a session
- How to use one thread per session
- How to share Fundamental JVI classes across threads

Multiple Threads, Multiple Sessions (MTMS)

In this model, no thread shares its session with another thread. In effect, all threads are in separate transactions and use different object caches. These threads can independently lock the accessed objects.

By default, the thread that creates a `FundSession` joins the session. If option `DONT_JOIN` is specified in the `FundSession` constructor, then the session starts without the thread joining the session.

To attach a thread to a session or switch from one session to another, invoke the `setSession()` method in that thread.

To detach the current thread from a session, invoke the `leaveSession()` method.

To end a session, invoke the `endSession()` method. This method by default requires the session not be associated with any other threads.

To end a `Session` that is in use by other threads, use the `ENDSESSION_FORCE` option.

To detach from a session in use by other threads, use the `ENDSESSION_DETACH` option.

You can invoke the `setDefaultSession()` to enable the automatic `setSession()` method to a chosen default `Session` object. You can invoke this method in applications where threads are created by underlying classes. For example, in GUI programs using AWT, event-handling threads are spawned by the AWT classes. If the event callback methods need to access persistent objects, then AWT event handling threads need to be attached to a session. Using the `setDefaultSession()` method saves the application from explicitly using `setSession()` in AWT event callback methods.

Multiple Threads, Single Session (MT1S)

In this model, multiple threads share a single session. This model also applies to an application that has multiple sessions with one of these sessions being shared by multiple threads.

MT1S is more difficult to use because the application must perform its own synchronization, in addition to the synchronization provided by the database.

How to attach threads to a session

To associate a thread with a session created in another thread, you need to explicitly join the session before working in that session.

When a thread joins a new session, it automatically leaves the session to which it is currently attached.

For example:

```
// ThreadX: Creates a Multi-thread Usable FundSession
FundSession session = new FundSession ("mydb@myserver");
// ThreadY: Joins a session instantiated in ThreadX.
session.setSession ();
```

How to detach threads from a session

When you do not need to use a previously joined session, you should explicitly leave it.

For example:

```
// ThreadY: Leaves a session joined earlier on
```

```
session.leaveSession ();
```

How to use one thread per session

The following is an example application that uses the one thread per session model: it has three threads and each thread has its own session. None of the threads uses more than one session, and no session is used by more than one thread.

This example is essentially the same as `StockMarket.java` located in ″How to use the MTMS mode″ in Transparent JVI. The source code of `FundStockMarket.java` can be found in your JVI installation at `example/fund/FundStockMarket.java`.

```
import com.versant.fund.*;
import java.util.*;

public class FundStockMarket
{
    public static void main (String[] args) throws InterruptedException
    {
        FundSession session = new FundSession("db", "main");

        AttrString symbol_attr = session.newAttrString("symbol");
        AttrInt    price_attr  = session.newAttrInt("price");
        AttrBuilder attrs[] = { session.newAttrBuilder(symbol_attr),
                                session.newAttrBuilder(price_attr) };

        ClassHandle cls = null;
        try {
            cls = session.withAttrBuilders (attrs).defineClass
                  ("StockQuote");
        } catch (VException vex) {
            if (vex.getErrno () == 6001) /* SCH_CLASS_DEFINED */ {
                cls = session.locateClass("StockQuote");
            }
            else throw vex;
        }

        for (int i = 0; i < symbols.length; i++) {
            Handle h = cls.makeObject ();
            h.put (symbol_attr, symbols[i]);
            h.put (price_attr, Math.abs(random.nextInt()) % 10000);
        }
        session.endSession ();

        Thread[] threads = {
            new StockMarketReader (),
            new StockMarketWriter ()
        };

        for (int i = 0; i < threads.length; i++) {
            threads [i].start ();
        }
        for (int i = 0; i < threads.length; i++) {
            threads [i].join ();
        }
    }

    static String randomSymbol ()
    {
        int index = Math.abs(random.nextInt()) % symbols.length;
        return symbols [index];
    }

    static boolean done = false;
    static Random random = new Random ();
 static String[] symbols = { "vsnt", "sunw", "ibm", "msft", "orcl" };
}

class StockMarketReader extends Thread
{
    public void run ()
    {
        FundSession session =    new FundSession("db", "reader");
        AttrString symbol_attr = session.newAttrString("symbol");
        AttrInt    price_attr  = session.newAttrInt("price");

        FundVQLQuery fvql = new FundVQLQuery(
                                 session,
                                 "select selfoid from StockQuote");
        while ( !FundStockMarket.done ) {
  HandleEnumeration he = fvql.execute(0, 0, Constants.IRLOCK,
                                                Constants.RLOCK);

            System.out.println ();
            System.out.println("Stock Market Report");
            while ( he.hasMoreHandles() ) {
                Handle quoteHandle = he.nextHandle ();
                System.out.println (quoteHandle.get (symbol_attr) +
                           ": $" +
                           quoteHandle.get (price_attr) / 100.0);
            }
            System.out.println ();
            session.commit ();

            try { sleep (1000); } catch (InterruptedException ex) {}
        }
        session.endSession ();
    }
}

class StockMarketWriter extends Thread
{
    public void run ()
    {
        FundSession session     = new FundSession ("db", "writer");
        AttrString  symbol_attr = session.newAttrString ("symbol");
        AttrInt     price_attr  = session.newAttrInt ("price");

        FundVQLQuery fvql = new FundVQLQuery (session,
           "select selfoid from StockQuote where symbol = $1");

        for (int i = 0; i < 200; i++) {
            String symbol = FundStockMarket.randomSymbol();
            fvql.bind (symbol);

            HandleEnumeration he = fvql.execute(0, 0,
            Constants.IWLOCK, Constants.WLOCK);
            Handle quoteHandle = he.nextHandle ();

            int delta = FundStockMarket.random.nextInt() % 200;
            quoteHandle.put(price_attr, quoteHandle.get (price_attr) +
                            delta);
            System.out.println ("Market update: " + symbol +
                                (delta >= 0  " up " : " down ") +
                                 Math.abs(delta) / 100.0);
            session.commit ();

            try { sleep (50); } catch (InterruptedException ex) {}
        }

        session.endSession ();
        FundStockMarket.done = true;
    }
}
```

How to share Fundamental JVI classes across threads

Instances of the following `com.versant.fund` Java classes should not be arbitrarily shared across threads. You must either use them only in the thread where instantiated, or you must explicitly manage your own synchronization when accessing them. This action is necessary to serialize thread calls to the methods on the following objects:

`Attr*` classes
`*Builder` classes
`Predicate`
`VException`
`HandleEnumeration`

The following classes can be shared across threads. All instances of these classes are immutable:

`NewSessionCapability`
`Handle`
`ClassHandle`

An instance of the `FundSession` class can be shared across threads. All of the methods of this instance are synchronized internally.

Multiple Databases

Overview

To explain multiple databases, this section describes:

- Multiple databases and sessions
- Multiple databases and methods

To explain how to use multiple databases, this section describes:

- How to connect with multiple databases
- How to disconnect multiple databases
- How to get schema from another database
- How to synchronize classes

Multiple Databases and Sessions

A session database is the database named in the session constructor. It cannot be changed once the session is started.

A session can connect multiple databases with the `connectDatabase()` method. The session can disconnect from a group database by calling the `disconnectDatabase()` method.

Multiple Databases and Methods

Methods on session objects operate on the session database, unless preceded by the `withDatabase()`method.

Methods on `ClassHandle` usually operate on the schema object in the session database. To work on a schema object in a connected database, use the `withDatabase()` method on the `ClassHandle` to specify the database.

You can use the `synchronizeClass()` method to synchronize the definition of a leaf class in a source database to a target database.

NOTE:- The `synchronizeClass()` method can drop all instances of the class in the target database when the `SC_DROP_INSTS` option is used.

How to connect with multiple databases

Fundamental JVI allows a session to be associated with multiple databases.

For example:

```
// connects to a database using the default access mode
session.connectDatabase ("myOtherDb");

// connects to a database using the specified
// read-only access mode:
session.connectDatabase ("myReadDb", Constants.READ_ACCESS);
```

How to disconnect multiple databases

You can disconnect from a connected database.

For example:

```
session.disconnectDatabase ("myOtherDb");
```

After disconnecting from a group database, any further attempts to use objects in that database are rejected.

How to get schema from another database

To access a class object in another group database, invoke the `withDatabase()` method on a `ClassHandle`. After the method invocation, the `ClassHandle` instance refers to the class object in the other database.

For example:

```
// cls refers to the class object in the session database
ClassHandle cls = session.locateClass ("MyClass");
// cls refers to the class object in the group database "otherDb"
cls = cls.withDatabase ("otherDb");
```

How to synchronize classes

You can duplicate the class definition of a leaf class (with no subclasses) from one database to another.

The following example attempts to make the definition of MyClass in database targetdb the same as in the session database. If targetdb already contains a different definition of class MyClass, all instances of MyClass will be dropped from targetdb.

For example:

```
ClassHandle cls = session.locateClass ("MyClass");
cls.synchronizeClass ("targetDb", Constants.SC_DROP_INSTS);
```

Memory Management

Overview

To explain memory management, this section describes:

- Object cache
- COD table
- COD

Object cache

Each session has an object cache in the C memory space to speed up access to objects in a transaction.

During a transaction, the object cache contains objects accessed in the current transaction.

When the application requests access to an object (example: using the Handle.get() method), Versant object manager first searches the object cache before it searches the database itself. Therefore, the first access to a persistent object in the transaction requires finding the object in the database, and then bringing the object into the object cache. Subsequent accesses to the object in the same transaction are much faster because the object is cached.

When a commit or roll back ends a transaction, the object cache is cleared.

COD table

The object cache maintains a cached object descriptor (COD) table of persistent objects. This table is a hash table keyed on LOIDs. Each entry of the COD table is a COD that corresponds to an object accessed in the current session.

When you commit or roll back a transaction, the object cache clears, but the COD table remains by default. Since the COD table is not cleared until the session is ended and if a large number of objects are accessed in a session, the COD table can potentially require much memory.

COD

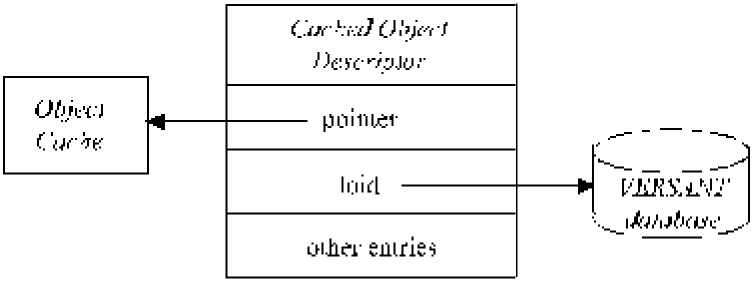
Each COD is associated with one persistent object accessed in the current session.

CODs are maintained regardless if the object is located in memory or on disk. Unless explicitly removed, a COD survives the ending of the transaction until the session is ended.

A COD can be explicitly removed from the COD table by zapping it.

See also "Using memory management APIs" on page 85.

The following shows the relationship of the COD, the object cache and the Versant database.



When the object cache clears, persistent objects marked as dirty are written to their database, and all persistent objects are dropped from object cache memory.

Handles and CODs

Each `Handle` corresponds to one COD entry, but many `Handles` can map to the same COD. Fundamental JVI maintains the many-to-one relationship.

`Handles` corresponding to a zapped COD should not be used. Instead, you should reconstruct a `Handle` based on the LOID of the zapped object.

Using memory management APIs

The following methods on `HandleVector` can help manage memory:

`releaseObjects()`

Release objects from the object cache, but retain their CODs in the COD table. Therefore a `Handle` is still usable after its object has been released.

A released object is not written to the database; instead, it is directly released from the object cache.

Releasing an object can be useful if you have no further use for the object in the current transaction. When the application requests access to the object again, the persistent object is fetched from its source database and placed in the object cache.

`zapObjects()`

Release objects and invalidate their CODs. `Handles` that correspond to this object should not be reused.

When the application needs access to a "zapped" object again, the application should reissue a query; the query causes a new COD to be constructed in the object cache and a new handle to be constructed for the new COD.

This method can be useful in a long-running session that accesses a large number of objects. You can selectively "zap" those objects no longer needed by the current session to control the growth of the COD table.

`groupWriteObjects()`

Options allow you to release objects and/or zap CODs.

The following method can also manage memory:

`commitAndCleanCod()`

Clear the COD table of entries of all objects.

Normally, the COD table is not cleared until the session ends. This method can be useful for those applications that populate databases. Such applications typically create many objects. However, once committed, the objects are not used further in subsequent transactions.

Any handles associated with the session should not be used after `commitAndCleanCod` is invoked.

How to get object cache usage

You can obtain information about the memory available in the front-end object cache.

For example:

```
int freeKB      = session.cacheFreeKb ();
int maxKB       = session.cacheFreeMaxKb ();
int percentFree = session.cacheFreePercent ();
```

Transaction Methods

commit()

`commit()` saves your actions to the associated databases and releases short locks. `commit()` also releases objects from cache memory, erases all savepoints, deletes objects marked for deletion and starts a new transaction. `commit()` makes no changes to transient objects.

If `Constants.RELEASE_SCHEMA_OBJ` is set while creating a session, then release the schema objects, else retain the schema objects in the objects cache.

Behavior of commit_delete

If the server profile parameter `commit_delete` in turned `OFF`, the delete APIs `deleteObject()` and `groupDeleteObjects()` write directly to the server. The objects are deleted immediately.

If `commit_delete` is set to `ON`, the delete APIs will mark the objects for deletion in the database server. The objects are immediately released from the front end cache. The server maintains a list of objects marked for deletion, and these are deleted when a commit is performed. In case of a rollback the objects are unmarked and they retain their original status.

If you desire a finer grade of control, you can use the following methods defined in `FundSession`:

`checkpointCommit()`

Use this method if you want to continue to use objects after the changes are committed to the database, and if you want to retain locks on these objects. This method writes new and dirty objects to the database, deletes objects marked for deletion, maintains short locks on all objects and maintains the object cache.

`commitAndRetain()`

Use this method if you want to continue to use objects after the changes are committed to the database. This method commits all new and dirty objects, deletes objects marked for deletion, releases locks held on all objects and retains persistent objects in the object cache. The method is defined on both `FundSession` and `HandleVector.`

`commitAndRetainSchema()`

Use this method when you know there will be no schema evolution. You can use this method to prevent releasing of the schema objects to improve performance. When instances of the involved classes are used later in the session, schema objects are released during a normal commit and roll back. Delete all objects marked for deletion.

`commitAndCleanCod()`

Use this method when memory is tight and you want to reclaim some memory by removing the CODs of objects no longer needed.

You can use this method to write new persistent objects to the default database, write changes of dirty objects to their databases, delete objects marked for deletion, release short locks, invalidate the object memory cache, clear the cached object descriptor table and start a new transaction.

See also “Using memory management APIs” on page 85.

`rollbackAndRetain()`

Use this method when you need to roll back objects, to unmark the objects marked for deletion, but retain the clean (not new or dirty) objects.

This method is defined on both `FundSession` and `HandleVector`.

Security

Overview

In some internet uses of Java, it is commonplace for some threads in the Java Virtual Machine to execute downloaded code automatically from the internet. For example, this download could be a side effect of visiting a web page.

From a security point of view, such code is termed untrusted and should be regarded as a potential adversary. Java's SecurityManager protects most operating system resources from untrusted code. However, the SecurityManager does not protect the native methods that give Java access to a Versant database.

To protect Versant databases, the `NewSessionCapability` class was created as part of Fundamental JVI. Its constructor enforces the rule that only one instance of `NewSessionCapability` can be created inside any Java Virtual Machine.

A single instance of `NewSessionCapability` must be created by trusted code, and a reference to it should only be given to other trusted code that can create new sessions.

If you want to create secure sessions, you must instantiate an instance of `NewSessionCapability` before you can begin a session with a Versant database. `NewSessionCapability` implements the `Capability` interface as follows:

```
Capability capability = new NewSessionCapability ();
```

If there is any untrusted Java code in your Java interpreter, the `NewSessionCapability` instance must be instantiated first with trusted code before any untrusted code runs.

Untrusted code should not be allowed to access the `NewSessionCapability` object. Since only one instance of `NewSessionCapability` can be created in each Java interpreter, any subsequent attempt to create an instance results in an exception.

After instantiation, every session that is created would need to use the single `NewSessionCapability` instance.

To explain how to set session security, this section describes how to create a secure session.

How to create a secure session

You can create secure sessions with a Versant database using an instance of `Capability`. The `Session` constructor accepts a `Capability` instance as a parameter.

The public class NewSessionCapability implements the Capability interface and its constructor enforces that only one instance can be created for each Java class loader.

A single instance of Capability must be created by trusted code. A reference to it should only be given to other trusted code that can create new sessions. When an instance of Capability is created, all other Session constructors need to use it.

For example:

```
// The Capability object is shared across threads
Capability cap = new NewSessionCapability ();

// ThreadA:
Properties prop = new Properties ();
prop.put ("database", "mydb@myserver");
FundSession session = new FundSession (prop, cap);

// ThreadB:
Properties prop = new Properties ();
prop.put ("database", "mydb@myserver");
FundSession session = new FundSession (prop, cap);
```

Alternately, an "insecure" mode can be used when instantiating sessions. A Session constructor that does not accept a Capability parameter can be used to instantiate such sessions.

For example:

```
// Thread1:
FundSession session = new FundSession ("mydb@myserver");
// Thread2:
FundSession session = new FundSession ("mydb@myserver");
```

When you create sessions without using a Capability argument, instantiation of a new NewSessionCapability leads to a VException being thrown.

Therefore, when you use Session constructors that do not use a Capability argument, you work in an "insecure" mode where any user thread can start a session on the database.

Optimistic locking

Overview

Standard Versant read, update and write locks provide consistent and coherent access guarantees in a multiple user environment. These guarantees are necessary and appropriate under most circumstances.

However, suppose you have the following scenarios:

- You want to read a large number of objects, but update only a few.
- You want objects that have a small chance of being updated by others. Holding a lock on all the objects you want could interfere with the work of others.

For situations such as these, Versant provides optimistic locking features that allow you to work with objects without holding locks.

You do not have to use optimistic locking features to safely access and update unlocked objects, but the side effects of the alternative approaches might not be right for your situation.

To increase concurrency of your application, you can use the optimistic locking features that allow you to work with objects while holding locks for shorter periods rather than using standard locking.

Candidates for using optimistic locking include applications that read a lot of objects but only modify a few of them, and applications that work on objects that other users are not likely to update.

Using optimistic locking

To use optimistic locking on an object, follow these steps:

1. Add a timestamp attribute to class definition
2. Begin an optimistic locking session
3. Follow the optimistic locking protocol

The above listed steps have been described in detail below:

Add a timestamp attribute to class definition

To make a class "optimistic lock ready", add a timestamp attribute to the classes that you want to use with the optimistic locking protocol.

In Fundamental binding, the timestamp attribute should be defined as follows:

```
AttrInt timestamp = session.newAttrInt("o_ts_timestamp");
```

In Transparent binding, the timestamp attribute should be defined as follows:

```
int o_ts_timestamp;
```

The commit, delete, group write and check timestamp methods will automatically use the timestamp attribute to determine if an object is an obsolete copy of the object in the database. An obsolete copy can occur if, while you are working with an unlocked copy of a persistent object, another user updates the object in the database.

Note that both an optimistic session and standard locking session will update the timestamp attribute. When an object with a timestamp is created in the database, its timestamp value is set to 0. Thereafter, each update written to a database will increment the timestamp value.

Begin an optimistic locking session

An optimistic locking session delays implicit lock upgrade to write lock (`WLOCK`) when you modify a persistent object until commit time, and therefore reduces the duration of holding of write locks in your application and increases concurrency. In a standard session, on the other hand, Versant attempts to acquire a write lock as soon as you modify a persistent object or mark the object as dirty.

An optimistic locking session also provides automatic detection of obsolete copies of objects with timestamp.

To start an optimistic locking session instead of a default standard session, use the `OPT_LK` option when you create a session object.

For example,

To start an optimistic locking session in the fundamental layer, you can create a `FundSession` object as follows:

```
Properties prop = new Properties ();
prop.put ("database", "mydb");
prop.put ("options", Constants.OPT_LK + "");
FundSession session = new FundSession (prop);
```

To start an optimistic locking session in the transparent layer, you can create a TransSession object as follows:

```
Properties prop = new Properties ();
prop.put ("database", "mydb");
prop.put ("options", Constants.OPT_LK + "");
TransSession session = new TransSession (prop);
```

Note that `OPT_LK` is an integer constant defined by class `Constants` in the Fundamental Package. In order to add it to a `Properties` object using the `Properties.put()` method, you need to convert the integer into a string first.

Follow the optimistic locking protocol

To ensure coherence and consistency normally provided by locks, follow the optimistic locking protocol summarized below.

For more information, please refer to "Locks" in the *Versant Database Fundamentals Manual.*

The protocol assumes that the objects involved have timestamp attributes and that the session used is an optimistic locking session.

a. Read objects with locks and pins

Fetch your "optimistic transaction set" from database with read locks (`RLOCK`) and pins. An optimistic transaction set refers to set of persistent objects that your application works on, together with objects inter-related with them.

Read locks are necessary to ensure consistency among objects in your optimistic transaction set. Pins are necessary to avoid automatic swapping in the Versant object cache.

Use `groupReadObjects()` method on a session object or `HandleVector` to bring a group of objects into the Versant object cache from the database and place a default lock on the objects. If the `PIN_OBJECTS` option is used in the group read method, objects will be pinned to suppress automatic swapping in the Versant object cache.

If an object is brought into memory by dereferencing, Versant places a default lock on the object, and you can use the `locateObject()` method on `Handle` to pin the object.

b. Downgrade short locks

Once you have a consistent set of objects, you can use the `downgradeLocksTo()` method on a session object or `HandleVector`, passing `NOLOCK` in the lockmode parameter, to drop locks without changing the pin status.

Alternatively, for a thread, you can turn on automatic lock downgrading using setThreadOptions() method with DROP_RLOCK as an argument.

c. Browse and update

Browse objects as usual. If additional objects outside of the optimistic set are fetched during browsing, a read lock may be placed by default as a side effect of dereferencing an object. To avoid setting locks, you can either set the default lock to NOLOCK or enable automatic dropping of read locks. However, it is better to avoid fetching additional objects so as not to introduce objects that may not be consistent with the original optimistic transaction set.

Updating persistent objects or marking them dirty in an optimistic locking session does not set a write lock on the objects immediately.

d. Check timestamps (preferred)

Before proceeding to a commit after modification, you can determine whether any of your changed objects have become obsolete by checking the timestamps of these objects.

The checkTimestamps() methods on a session object and HandleVector check all dirty objects in a group of objects; checkAllTimestamps() on FundSession (and its subclasses) checks all dirty objects in the Versant object cache.

The check timestamp methods set the default lock on all objects checked. If obsolete objects are found, an instance of VException (4036, OM_TR_INVALID_TS: Failed time stamp validation) is thrown. You may want to refresh your copy with the refreshObjects() method defined in TransSession or HandleVector.

If the server profile parameter commit_delete is set for a database, the delete operation is delayed until commit. If some objects are "Marked for deletion" by some other application, an error SM_LOCK_TIMEDOUT will be returned. The application that marks these objects for deletion acquires a Write lock on them, and hence checkTimestamps() or checkAllTimestamps() will be unable to acquire a lock on them.

e. Commit or rollback changes

You can save your changes with a transaction commit or undo your changes with a transaction rollback. Both commit and rollback end the transaction and start a new transaction.

Note that some commit/rollback methods retain objects in the object cache so that you can continue working on them and some methods flush the object cache. Some commit/rollback methods only commit or rollback a group of objects specified by you and some scan all objects in the object cache for dirty objects.

NOTE:-

- If the server profile parameter commit_delete is OFF, this function will send the delete request to the source database and the object is deleted immediately.

- If `commit_delete` is `ON`, this function will acquire a Write Lock on the object and set its status as "`Marked for deletion`". The object will be physically deleted at commit.

  If the transaction commits, the objects are physically deleted. If a rollback occurs, these objects are un-marked and their status restored.
- Queries run on the database will not include objects marked for deletion in the result sets.

During commit time, Versant attempts the following in sequence:

- Increment the timestamp by 1 of each dirty object with a timestamp attribute in the object cache.
- Set a write lock on the source object of each dirty object in the database.
- Compare the timestamp value of each dirty object in the object cache with its source object in the database.
- Write the dirty object to the database if the timestamp of the dirty object is greater than that of the source object by 1. Otherwise, the dirty object is not written.

If one or more dirty objects fail the timestamp validation, commit fails and an `VException` is thrown to signal the timestamp validation failure (4036, `OM_TR_INVALID_TS`: Failed time stamp validation).

If commit fails, either because of timestamp validation failure or other errors, you can refresh the failed objects to load the latest committed copy from the database then decide whether to re-apply your changes or not.

f. Possibly recover from exceptions

If commit, check timestamp or group delete fails, use the `getFailedObjects()` method to retrieve the obsolete objects in an array if you are using `TransSession` in the transparent package or `getFailedHandles()` to retrieve an array of handles to the failed objects if you are using `FundSession` in the Fundamental Package.

The format of the array of failed objects is as follows:

```
{ failed_timestamp, ..., NULL, deleted, ..., NULL }
```

where "NULL" is the null object for `getFailedObjects()`, and the empty handle (handle to 0.0.0) for `getFailedHandles()`. "NULL" is as a delimiter to separate the group of objects that failed the timestamp check from the group of objects that have been deleted.

Schema Evolution

Overview

Versant stores the class definition of all used classes on the database. This stored definition (schema) must be identical to the class definition in your program to avoid unpredictable results. In addition, when you change a class definition, existing instances of the changed class must adapt to the new definition.

To change the definitions, use the following methods defined in `ClassHandle`:

Method	Description
`renameAttr`	Changes the name of an attribute in the class.
`renameClass`	Changes the name of the class in the source database.
`dropAttr`	Removes the attribute name from the source database.
`appendAttr`	Adds the new attribute at the end of the current set of attributes in the class in the source database.
`insertAttrsAt`	Adds the new attributes to the class in the source database at the specified position.

NOTE:- These methods attempt to acquire write lock on the specific class object.

You can use `ClassHandle` to modify class definitions by:

- Adding an attribute
- Dropping an attribute
- Renaming an attribute
- Renaming a class

How to add an attribute

You can add a new attribute to the current set of attributes in a class.

The following example appends a new attribute named `"description"` of type `string` to class named `"Report"`.

For example:

```
ClassHandle cls  = session.locateClass ("Report");
```

```
AttrString  desc = session.newAttrString ("description");
AttrBuilder attr = session.newAttrBuilder (desc);
cls.appendAttr (attr);
```

How to drop an attribute

You can drop an existing attribute from a class.

For example:

```
ClassHandle cls = session.locateClass ("Report");
cls.dropAttr ("attr");
```

How to rename an attribute

You can rename an existing attribute in a class.

For example:

```
ClassHandle cls = session.locateClass ("Report");
// rename "description" to "subject"
cls.renameAttr ("description", "subject");
```

How to rename a class

You can rename the class itself.

For example:

```
ClassHandle cls = session.locateClass ("Report");
// Renaming "Report" to "Form"
cls.renameClass ("Form");
```

Object Migration

The ability to migrate objects makes distributed databases practical. After an object has been created, you could migrate it and place it physically closer to where it is most often used. This action will reduce network traffic. If the disk space is full on the machine that contains the creation database, you could migrate the object also.

Object and database identifiers do not change when an object or database is moved or changed. Identifiers are never reused, even after an object or database has been deleted. Therefore, the code that references an object does not have to be changed each time an object is migrated.

Each persistent object must have schema objects associated with it that contains its class definition. When you migrate an object to a new database that is not already defined in the target database, a copy of its schema objects is also migrated.

Since migrating an object instance includes migration of its definition in schema objects, a particular set of schema objects could be duplicated in numerous databases. However, once a schema object has been migrated, it exists as an independent object in that database. Therefore, to use a migrated object, you do not have to be connected to its creation database.

If classes of the same name are defined differently in different databases, an attempt to migrate an object with a different class definition is automatically blocked. You can then use any of a variety of mechanisms, such as synchronizing the class to resolve the differing class definitions.

The `HandleVector` class contains the `migrateObjects()` method to migrate a group of objects from the source database to the target database. It also contains `copyObjects()` method, which differs from `migrateObjects()` only in that the former leaves the involved objects in the source database, whereas the latter does not. Copying objects can serve as a mechanism to replicate selected objects programmatically.

Group Operations

A HandleVector represents a collection of Handles. The HandleVector class extends java.util.Vector, so all the standard operations that are possible with a Vector can be performed on a HandleVector.

Accessing elements

You can use HandleVector to access elements by:

- Adding elements
- Setting a Handle in the HandleVector
- Getting a HandleEnumeration from a HandleVector.
- Iterating HandleEnumeration and accessing individual elements.

Adding elements

You can add elements to a HandleVector.

For example:

```
HandleVector hv = new HandleVector ();
hv.addHandle (handle);
```

Setting a Handle in the HandleVector

You can set the Handle at a specific position in the HandleVector.

For example:

```
hv.setHandleAt (handle, 10);
```

Getting a HandleEnumeration from a HandleVector

The HandleEnumeration class implements the java.util.Enumeration interface. You can use this class to iterate through a collection of Handles in the standard way with an Enumeration.

For example:

```
HandleEnumeration e = hv.handles ();
```

Iterating HandleEnumeration and accessing individual element

You can iterate through a `HandleEnumeration` and access individual elements.

For example:

```
HandleEnumeration e = hv.handles();
while ( he.hasMoreHandles() ) {
   Handle handle = e.nextHandle ();
   System.out.println (handle);
}
```

Performing group operations

You can use `HandleVector` to perform database group operations.

Group operations are generally faster than operating on individual objects for multiple times.

To explain how to use HandleVector to perform group operations, this section describes:

- How to read objects as a group
- How to write a group of objects
- How to delete a group of objects
- How to migrate a group of objects
- How to copy a group of objects
- How to update a group of objects in front-end cache
- How to release a group of objects from the object cache
- How to zap a group of objects from the object cache

How to read objects as a group

You can read objects as a group. Operations on a group of persistent objects allow your application to perform an operation using a single database call. This single database call can provide a performance benefit to an application. For example, following a query, a `groupReadObjects()` method places all the read objects in the front-end object cache at once.

For example:

```
hv.groupReadObjects ("myDb@myServer", 0, Constants.RLOCK);
```

How to write objects as a group

You can write a group of objects to the database and release their entries in the COD table for memory management purposes.

For example:

```
hv.groupWriteObjects ("myDb@myServer", Constants.CLEAN_CODS);
```

How to delete objects as a group

You can delete a group of objects at once rather than delete the objects one at a time and incur additional network overhead for each individual method call.

NOTE:-

- If the server profile parameter `commit_delete` is `OFF`, this function will send the delete request to the source database and the object is deleted immediately.
- If `commit_delete` is `ON`, this function will acquire a Write Lock on the object and set its status as "`Marked for deletion`". The object will be physically deleted at commit.

  If the transaction commits, the objects are physically deleted. If a rollback occurs, these objects are un-marked and their status restored.
- Queries run on the database will not include objects marked for deletion in the result sets.

For example:

```
try {
   hv.groupDeleteObjects ("myDb@myServer");
} catch (VException vex) {
   // if some objects could not be deleted
   HandleVector failedVector = vex.getFailedHandles ();
}
```

How to migrate objects as a group

You can migrate a group of persistent objects from a source to a target database.

For example:

```
try {
   // Migrate from myDb (source) - newDb (target)
   hv.migrateObjects ("myDb@myServer", "newDb@myServer");
} catch (VException vex) {
   // if some objects could not be migrated
   HandleVector failedVector = vex.getFailedHandles ();
}
```

How to copy a group of objects

You can copy a group of persistent objects from a source to a target database.

For example:

```
try {
   // Copy from myDb (source) - newDb (target)
   hv.copyObjects ("myDb@myServer", "newDb@myServer");
} catch (VException vex) {
   // if some objects could not be migrated
   HandleVector failedVector = vex.getFailedHandles ();
}
```

NOTE:- Transparent JVI (`com.versant.Enhance`, `com.versant.DefineClass`, `com.versant.Launch`) has its own mechanism to order attributes in database whereas using Fundamental JVI, the user can influence order in database by using `AttrBuilder[]`, `dropAttr()`, `renameAttr()`, `insertAttrsAt()`.

The API's, `copyObject()` and `migrateObjects()` work only if schema in source and target database are identical. They will fail if order of attributes are different despite the fact that attributes and their domains are equal.
So if the database schema is modified using Fundamental JVI APIs, then `copyObject()` and `migrateObjects()` will throw error `E6063:SCH_MIGR_NO_VALID_OBJS`.

How to update a group of objects in the object cache

You can update a group of persistent objects in the front-end object cache with the current state in the database; pin the objects in the cache and set a given lock mode on all the objects.

For example:

```
try {
   // Each object in the handle vector is reread from the
   // database, with a write lock placed on the object
   hv.refreshObjects ("myDb@myServer", Constants.WLOCK);
} catch (VException vex) {
   // if some objects could not be refreshed
   HandleVector failedVector = vex.getFailedHandles();
}
```

How to release a group of objects from the object cache

You can use the `releaseObject()` method on the `Handle` to release an object from the front-end object cache. This method releases a previously pinned or dirtied persistent object. The `releaseObject()` method is also typically used when there is no further use for an object in a transaction. If you want to write the changes to the database, you must write out the dirty objects before you release them.

For example:

```
// Release all objects in the handle vector
hv.releaseObjects (0);
```

How to zap a group of objects from the object cache

You can release a specified group of objects from the front-end object cache and make their cached object descriptors available for reuse with the `zapObjects()` method. After invoking this method, you should not use the `Handles` in the `HandleVector` in the remainder of your application.

For example:

```
// Zap the COD entries for all objects in the handle vector
hv.zapObjects ();
```

CHAPTER 3

Transparent JVI

This Chapter explains the concepts of “Transparent JVI”.

The Chapter covers the following in detail:

- Introduction
- Basic Concepts
- Advanced Concepts
- Tutorial for Transparent JVI

INTRODUCTION

Overview

The JVI Transparent binding unifies the Java language with the Versant object model. It allows applications to be written in a very natural way, where some of the Java objects are completely persistent and transactional. Persistent Java classes do not use any special syntax or conventions; instead you declare classes to be persistent using a special tool, the Enhancer. Your Java program can use a persistent object just like any other Java object. For example, you can call its methods; read and write its fields, including object references and collections; and let other, possibly persistent, objects reference it.

The Versant ODBMS

The Transparent binding seamlessly inter-operates with the Versant object database. Persistent objects are automatically fetched and locked as your application accesses them, by traversing object references or using the Versant query mechanism. All access to persistent objects occurs within a Versant transaction, so that changes to an object can be atomically committed or rolled back.

JVI automatically maps the classes and fields of persistent Java objects to the Versant object model. Most types of fields correspond naturally to Versant attribute types; others will be converted to byte streams using standard Java serialization.

Caching Persistent Objects

Persistent objects created with the Transparent binding are cached in Java memory. When a persistent object is first accessed, it is fetched from the database, and a corresponding Java object is constructed. Contrast this with the Fundamental binding which, like the Versant C and C++ interfaces, caches persistent objects in C memory.

Since persistent objects are cached in Java memory, the garbage collector can reclaim the memory from objects that are no longer referenced by the application. Note that the garbage collector only removes objects from memory, not from the database. An object is not removed from the database without an explicit delete operation.

Relationship between Transparent JVI and Fundamental JVI

The following describes the relationship between Transparent JVI and Fundamental JVI:

- Transparent JVI is layered on top of Fundamental JVI.
- Not all Versant APIs are exposed directly in Transparent JVI. You can use Fundamental JVI to provide better access to Versant ODBMS . For example, if you want to create an index on an attribute, you could use Fundamental JVI instead of Transparent JVI because it provides direct access to Versant ODBMS.
- Persistent objects in Transparent JVI can be converted to `Handles` in Fundamental JVI.
- `Handles` in Fundamental JVI can be converted to persistent objects in Transparent JVI.

What's in this chapter?

This chapter contains explanations of specific basic and advanced concepts and shows you how to use classes in the Transparent JVI package based on the concepts.

Concepts and Tasks

The following lists basic concepts and associated tasks:

Basic Concepts	**Tasks**
Sessions	Explanation only
First Class Objects	How to create persistent objects with FACs
Enhancer	Explanation only
Persistence Categorization	How to choose a persistence category
Transparency	• How to fetch persistent objects • How to write changes to persistent objects • How to lock persistent objects • How to use transitive persistence
Cache Iterator	How to iterate over cached objects
Second Class Objects	• How to store Scows in persistent objects • How to use SCO tracking

Queries	• How to use predicate objects • How to use VQL queries • How to define roots • How to find roots • How to delete roots
Error Handling	How to catch exceptions when accessing an array of persistent objects

Advanced Transparent JVI Concepts and Tasks

The following lists advanced concepts and associated tasks:

Advanced Concepts	**Tasks**
Schema Objects	• How to use the DefineClass utility • How to evolve database schemas
Threads and Multiple Sessions	• How to use the MTMS model • How to use the MT1S model
Life Cycle of a Persistent Object	Explanation only
Object Identity	Explanation only
Serialization	Explanation only
Cloning	Explanation only
Transient Fields	Explanation only
Persistent Object Hooks	How to use the Persistent Object Hooks
Class Loaders	How to use class loaders with JVI
Object Sharing	Explanation only
Restrictions	Explanation only
Categorization of Interfaces and Superclasses	Explanation only
Performance Tuning	Explanation only
Group Operations	Explanation only

BASIC CONCEPTS

Sessions

Overview

You can connect to the Versant ODBMS in a single Versant session or in multiple concurrent sessions.

To explain sessions, this section describes:

- The relationship between a session and a transaction
- Sessions and the object cache
- Session classes and the session interface

Relationship between a session and a transaction

Please refer to Section *"*Relationship between a session and a transaction*"* in Fundamental JVI*.*

Sessions and the object cache

Each session maintains an object cache of persistent objects. In Transparent JVI, Java memory maintains the object cache.

Objects can be automatically removed from the cache by the garbage collector when they are no longer referenced in the application.

You can explicitly control the cache for better performance and/or memory utilization.

Session classes and the session interface

The `Session` interface is defined in the Fundamental JVI package, `com.versant.fund`.

`TransSession` is a concrete class defined in the Transparent JVI package. `TransSession` implements the `Session` interface. Most database-related APIs are located in the `TransSession` class.

First Class Objects

Overview

Persistent objects in the database are accessed as corresponding Java objects. These Java objects are called First Class Objects (FCO). The objects stored in the database closely mirror these First Class Objects. Attributes of the FCO correspond to attributes of the persistent database object in the following manner:

- Elemental types in Java such as `int` and `double` map naturally to Versant attribute types (example: `o_4b and o_double`).
- Java references to other FCOs map to Versant links.
- Java types that do not map directly to Versant types are serialized into a stream of bytes, using the standard Java serialization mechanism.
- If the Java class extends another Java class, the database object has a corresponding superclass.

The following describes how to create FCOs and how to store the objects in the database.

How to create persistent objects with FCOs

Consider the following simple `Person` and `Student` Java classes. Storing instances of these classes in the database is very straightforward. Note that there is no special syntax necessary when defining the classes.

```
class Person
{
  String name;
  int age;
}
class Student extends Person
{
  double gpa;
  Course[] courses;
  char[] grades;
}
```

To create a persistent `Student` object, simply construct a new object and call the `TransSession.makePersistent()` method. (This example assumes that the `Person`, `Student`, and `Course` classes have been categorized as Persistent Capable.)

See also “Persistence Categorization” on page 117.

For example:

```
Student student = new Student ();
student.name  = "John Q. Student";
student.age   = 22;
student.gpa   = 3.4;
student.courses = new Course [5];
student.grades = new char  [5];
session.makePersistent (student);
```

When the transaction associated with this session commits, the `Student` object is stored in the database. If this is also the first time that a `Student` or `Person` class is made persistent, then the schemas for these classes are defined automatically.

For more control over the definition of schema objects, use the DefineClass utility or the `TransSession.defineClass()` method, please refer to Section “Schema Objects” on page 156.

The schemas of these two classes look as follows (as reported by the `db2tty` utility):

- A special superclass, `CapableWithHash`, and a special attribute, `_vj_hashCode`, have been added. These support persistent hash codes.

See also “Object Identity” on page 180.

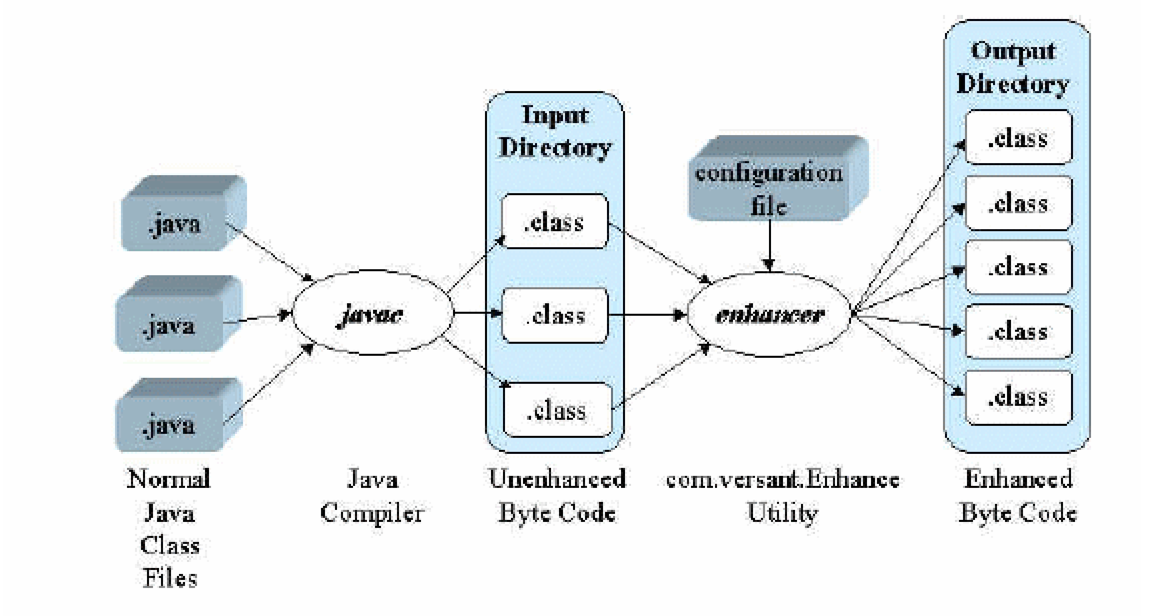
For example:

```
====== CLASS `Person' ======
superclasses:
 `com.versant.trans.CapableWithHash'
attributes:
 _vj_hashCode   : o_4b
 age        : o_4b
 name       : char[]
====== CLASS `Student' ======
superclasses:
 `Person'
attributes:
 _vj_hashCode     : o_4b
 age            : o_4b
 name           : char[]
 grades          : o_2b[]
 courses          -> (NULL_DOMAIN)[]
 gpa            : o_double
```

Enhancer

Overview

The enhancer is a tool that "post-processes" Java class files by modifying the classes after they are compiled. The enhancer accepts Java class files as input and produces new and modified class files as output. Transparent JVI uses the enhancer to achieve transparent FCO mapping to database objects.



The enhancer makes several changes to the class files, depending on the specifications in a configuration file. For an explanation of this configuration file, see the "Persistence Categorization" section below. These class file changes include:

- Code is generated to define the schema for the database objects.

  A schema object is created in the database when the first instance of a class becomes persistent. Note that the schema object can also be created explicitly. See the *"Schema Objects"* section in this chapter.
- Code is generated to read and write the persistent objects to and from the database.
- Code that accesses (gets and puts) fields of an FCO is modified to first perform additional checks.
- When a field of an FCO is read or modified, the object is fetched from the database as necessary.
- When a field of an FCO is modified, the object is marked "dirty." Therefore, changes are written to the database when the transaction is committed.
- The FCOs are "re-rooted" to extend a common base class. This base class is an internal detail of the implementation of Transparent JVI and should not have a direct effect.

Running the Enhancer

The enhancer can be invoked in two different ways: as a separate standalone utility, and as a Java class loader. These two mechanisms are described below.

Standalone Enhancer Utility

You can explicitly enhance your application classes using the standalone enhancer utility. This utility is the Java application `com.versant.Enhance`. When this application is executed, it does the following:

- Read the configuration file specified on the command line
- Read each class file (or collection of class files within a directory) specified on the command line
- Enhance each class file, using the information given in the configuration file

The enhancer can place modified class files in a separate directory, or it enhances class files "in-place." The optional `-out` flag is used to tell the enhancer where to place the modified class files. When using the `-out` option, JVI applications are typically structured with separate "in" and "out" directories. For example, consider an application whose classes are contained in the mycorp.myapp package. The directory structure might look as follows:

```
myrootdir/myconfigfile
myrootdir/in/mycorp/myapp/Main.java
myrootdir/in/mycorp/myapp/Main.class
myrootdir/in/mycorp/myapp/MyObject.java
myrootdir/in/mycorp/myapp/MyObject.class
myrootdir/out/
```

Invoke the enhancer utility as follows:

```
cd myrootdir/
java com.versant.Enhance -in in -out out -config myconfigfile
```

These arguments instruct the Enhance application to look for classes in the `in` directory, modify them according to the settings in `myconfigfile`, and deposit them in the `out` directory. This will cause the following directories and files to be created:

```
myrootdir/out/mycorp/myapp/Main.class
myrootdir/out/mycorp/myapp/MyObject.class
myrootdir/out/mycorp/myapp/MyObject_Pickler_Vj.class
```

Note the newly created "`MyObject_Pickler_Vj`" class. Creation of this class depends on the configuration file (in fact, the pickler class will only be created if the class is categorized as "c" or "p").

See also “Persistence Categorization” on page 117.

Alternatively, class files can be modified "in-place." This means that the enhanced class files replace the original class files. To use in-place enhancement, omit the `–out` flag. When using in-place enhancement, separate "in" and "out" directories are not necessary. For instance, the directory structure for the `mycorp.myapp` would simply be as follows:

```
myrootdir/myconfigfile
myrootdir/mycorp/myapp/Main.java
myrootdir/mycorp/myapp/Main.class
myrootdir/mycorp/myapp/MyObject.java
myrootdir/mycorp/myapp/MyObject.class
```

Then the enhancer can be invoked in the following manner:

```
cd myrootdir/
java com.versant.Enhance -config myconfigfile .
```

Note the "." at the end of the command line. This tells the enhancer to examine all classes in the current directory.

It is not possible to enhance a class more than once. The enhancer places a special marker inside the class file, so that it can determine whether a class has already been enhanced. If a class file has already been enhanced, then it will not be modified.

WARNING:- A persistent class is changed in such a way that other (persistent aware or persistent capable) classes that refer to the fields of the persistent class will no longer compile correctly when using in-place enhancement. Hence it is necessary to compile all of these classes simultaneously to ensure correct compilation. As an example, consider the following classes:

```
class PersistentCapable {
  int x;
}
class PersistentAware {
  int foo (PersistentCapable pc) {
   return pc.x;
  }
}
```

The following would not work correctly:

```
javac PersistentCapable.java
java com.versant.Enhance PersistentCapable.class
javac PersistentAware.java
```

Here, the PersistentCapable class has been enhanced in-place. Now, when PersistentAware is compiled, a compilation error will result because the field "x" has been changed to private. (The access control is changed to prevent unenhanced persistent aware classes from accidentally accessing the fields of a persistent object at run-time. By making the field private, this situation can result in an IllegalAccessError instead of silently giving invalid results.)

Instead, you should do this:

```
javac PersistentCapable.java PersistentAware.java
java com.versant.Enhance PersistentCapable.class \ Persisten
tAware.class
```

Enhancer Dependency on the Java CLASSPATH

Before invoking the Enhancer tool, the CLASSPATH must be set so that the enhancer can load the class being enhanced and its super classes.

There are two situations to consider here:

1. The out (target directory) and in (source directory) directories are the same (in-place enhancement)
2. The out and in directories are distinct.

If in-place enhancement is being done then the CLASSPATH should be set such that classes in the enhancement directory are picked up. For example: If the enhancement directory is "`myrootdir/in`", then the CLASSPATH should be set so that the classes are loaded from "`myrootdir/in`" directory. Typically if in-place enhancement is used, the CLASSPATH setting is the same as the application runtime CLASSPATH.

If in-place enhancement is not being done and the source and target directories are different, then the CLASSPATH should be set to the source directory before running the enhancer.

NOTE:- After Enhancement, the application runtime CLASSPATH will have to be changed so that the enhanced classes are picked up from the target directory.

Sufficient attention should be paid to setting the CLASSPATH correctly. If the class or the superclasses for the class being enhanced cannot be loaded into the JVM based on the CLASSPATH setting then the enhancer tools terminates with the exception `CannotFindClassException`.

If the CLASSPATH is set, but set incorrectly then, its possible that the Enhancer reflects on the wrong class to determine the declaring class of a field. Consider the following example:

Directory '`mydirectory/in'` has `class Person` that needs to be enhanced. `class Person` is declared as follows:

```
class Person {
  int id;
}
```

Let us assume there is another directory `'wrongdirectory'` that also has a `class Person` declared but whose declaration is as follows:

```
class Person {
  String name;
} String address;
```

If the CLASSPATH is set such that the `class Person` in *'*`wrongdirectory`*'* is loaded instead of class Person in *'*`mydirectory/in`*'*, then the enhancer will terminate with a `FieldNotFoundException` indicating that it could not find the field *'*`id`*'* in class Person.

As mentioned before, the Enhancer uses reflection to determine the declaring class for fields in the class being enhanced. The enhancer will terminate with a `SecurityViolationException` exception if the security manager denies access to the class or field.

For other information, on the enhancer, including command-line arguments and the syntax of the configuration file, please refer to description of `com.versant.Enhance` in the *JVI javadoc*.

JVI Enhancing Class Loader

The JVI enhancing class loader provides a more transparent alternative to the standalone utility. The class loader dynamically enhances Java classes "on the fly" while the classes are loaded.

This has an important advantage over the standalone enhancer: there is no separate enhancement step after compilation. You can simply compile and run your application directly; the necessary class enhancements will occur as the application runs. This makes it easier to use in integrated development environments.

Of course, enhancing classes at runtime adds to the overall execution time of your application, so you may not want to use the class loader when the application is deployed.

To use the enhancing class loader, use the `com.versant.Launch` utility to start your application. This puts the JVI class loader in place, and invokes the `main` method of your application.

For example, the `mycorp.myapp` application discussed above could be enhanced and executed simultaneously using the following command:

```
cd myrootdir/in/
java com.versant.Launch –config ../myconfigfile –out ../out \
            mycorp.myapp.Main arg1 arg2 ...
```

For more information, on the class loader, including command-line arguments, please refer to the description `com.versant.Launch` in the *JVI javadoc*.

Persistence Categorization

Overview

Each class in Transparent JVI has a persistence category. A persistence category describes the database-related capabilities of instances of a class. For example, can objects be stored in the database as FCOs? If so, when do the objects become persistent? Can the objects access FCOs? A persistence category also determines the modifications the enhancer must make to the class file for a class.

A configuration file is used to specify the persistence category for each class. Each line of the configuration file contains a category specifier and a class name pattern. When the enhancer encounters a class, it looks for the first matching pattern in the configuration file and gives the class the category specified for that pattern.

Transparent JVI has five persistence categories. The following describes each category:

Persistence Category	**Description**
"c", Persistent Capable	Objects of classes that are Persistent Capable can become persistent either explicitly or implicitly. Persistent Capable objects that have not yet become persistent are called transient.
"p", Persistent Always	Objects of classes that are Persistent Always become persistent when the object is created.
"d", Transparent Dirty Owner	Objects of classes that are Transparent Dirty Owner are embedded in FCOs. Transparent Dirty Owner objects perform SCO tracking. **See also "SCO tracking" on page 129.**

"a", Persistent Aware

Classes that are Persistent Aware have their methods modified by the enhancer in a way that allows them to correctly interact with FCOs.

Any accesses to the fields of an FCO are changed so that the object is automatically locked and fetched from the database if necessary.

Assigning new values to the fields of an FCO causes the object to be exclusively locked and marked as dirty. This assignment enables changes to be written back to the database when the transaction is committed.

"n", Not Persistent Aware

The enhancer does not modify classes that are Not Persistent Aware.

The methods of Not Persistent Aware classes cannot directly access the fields of an FCO.

The methods of Not Persistent Aware classes can call methods on an FCO.

How to Choose a Persistence Category

Choosing the correct persistence category for each of the classes in your Transparent JVI application is an important part of the development process. To properly categorize your classes, it is helpful to have understanding of how the classes will function within your application, as well as what changes the enhancer will make to these classes.

Persistent Capable

You should assign a class the Persistent Capable category if you want instances of that class to be stored in the database as independent persistent objects. A Persistent Capable object can be either transient or persistent. A transient object behaves essentially as it would have before the enhancer modifies the class. A persistent object, however, is a First Class Object. It takes advantage of the changes made by the enhancer to enable the object to be stored in the database. For example:

- The Versant database assigns each persistent object a unique object identity, its LOID.
- The enhancer generates additional methods that automatically read and write the object when the attributes of the object are read or changed.

For more information, on the changes made by the enhancer for Persistent Capable classes, please refer to Section below on "Transparency".

Persistent Always

The Persistent Always category is a minor variation of Persistent Capable. It simply means that any instances of a Persistent Always class becomes persistent as soon as it is created. That is, the constructor of a Persistent Always class automatically invokes the `makePersistent` method, causing the object to immediately become persistent.

Transparent Dirty Owner

Use the Transparent Dirty Owner category to embed an object (or network of objects) within another persistent object. The objects are embedded using the Java serialization mechanism, which turns them into a stream of bytes. Therefore the Transparent Dirty Class must be serializable for its instances to be embedded in a persistent object. The persistent object that contains the Transparent Dirty Owner object stores the stream of bytes.

The enhancer modifies a Transparent Dirty Owner class so that it performs "SCO Tracking". SCO Tracking means automatically marking a persistent object as dirty when an embedded object is changed.

See also “SCO tracking” on page 129.

Embedding an object is useful when the object does not need to maintain a separate identity in the database. The embedded object is logically contained within another persistent object.

Strings and arrays are simple examples of objects that can be embedded within a persistent object. These objects, however, map naturally to attributes of a Versant object. Transparent Dirty Owner extends this notion to arbitrary serializable types.

Embedded objects are also called Second Class Objects.

See also “Second Class Objects” on page 243.

Persistent Aware

A Persistent Aware object is modified so that it "knows" about the changes the enhancer makes to a Persistent object. Essentially this means that certain operations, such as accessing the fields of a First Class Object, are enhanced so that the persistent objects are automatically read or written from the database and assigned the appropriate lock modes.

You should assign the Persistent Aware category to a class if that class accesses any of the fields of a persistent object. A Persistent Aware class should behave correctly when using either persistent or non-persistent objects. Thus choosing the Persistent Aware category instead of Not Persistent Aware (see below) may incur additional runtime overhead, but it should not cause incorrect behavior.

Not Persistent Aware

The enhancer does not modify a Not Persistent Aware class. Classes that do not directly manipulate persistent objects can be categorized as Not Persistent Aware. For example, all standard Java classes are categorized as Not Persistent Aware by default. In general, third-party classes that do not use persistent objects can be given the "n" category.

If a Not Persistent Aware class tries to access the fields of a persistent object, it will throw an `IllegalAccessError` exception. See the *“Standalone Enhancer Utility”* section above.

Transparency

"Transparency" allows your Java application to see all objects as standard Java objects and manipulate them with standard Java methods.

To achieve transparency, JVI uses the enhancer, as described in the *"Enhancer"* section of this chapter. The enhancer takes as input a Java class file with Java byte codes. The Java byte codes are output from a Java compiler. The post-processor enhances these byte codes with additional byte codes. The additional byte codes enable instances of the Java class to become persistent.

The enhancement to the class adapts it to the requirements for a persistent object.

Implicit fetching and writing

Attributes of a persistent First Class Object are accessed in the same way as the attributes of a regular Java object. The first time the fields of an FCO are accessed in a transaction, the object is automatically fetched from the database. When the object is fetched, the fields are updated in the FCO.

If an FCO is modified during a transaction, the object is automatically marked as dirty and written to the database when the transaction is committed.

The following describes how you can implicitly fetch and write changes to persistent objects.

How to fetch persistent objects

Objects are fetched implicitly by simply accessing one of its fields.

For example:

```
boolean checkLegalAge (Person person)
{
  return person.age >= 21;
}
```

If the `checkLegalAge()` method is invoked after beginning a transaction, then the `Person` object is fetched from the database when the `person.age` expression is evaluated. This call also causes the object to be locked with the "default lock," which is normally an `RLOCK`.

See also "Implicit locking" on page 121.

How to write changes to persistent objects

Any change to a persistent object is automatically written to the database when the associated transaction commits.

For example:

```
void setName (Person person, String name)
{
 person.name = name;
}
```

After the `setName()` method is invoked, the `Person` object is marked as dirty, and the object is written to the database when the associated transaction commits. Modifying the object also causes the object to be locked with a `WLOCK`.

See also "Implicit locking" on page 121.

Implicit locking

The first time the fields of a First Class Object are accessed in a transaction, the object is automatically locked. Recall that all persistent objects in memory are associated with a Versant session, and that each session manages a single transaction. Therefore an FCO is locked in the transaction of the associated session.

If the field is being read, the default lock is obtained on the FCO. This default is typically a read lock. If the field is being set, then a write lock is acquired on the object.

Since JVI will automatically lock persistent objects as they are accessed, it is easy to forget that objects are locked at all. However, it is important to be aware of this locking, since locking-related problems such as deadlock can occur if care is not exercised. An example of this problem is given in the next section on "Explicit Locking."

Explicit locking

Consider what happens when an object is first read. The object is locked using the default lock and subsequently modified. The modification sets a WLOCK. Now assume that the default lock is RLOCK, as usual. If this is the case, then RLOCK on the object is upgraded to a WLOCK on modification. If multiple threads (or processes), associated with separate sessions (and hence separate transactions), perform this operation on the same object, then deadlock can occur.

For example, suppose the StockQuote class below is Persistent Capable.

```
class StockQuote
{
  String symbol;
  double currentPrice;
  double highestPrice;

  void setPrice (double price)
  {
   if (price > highestPrice) // sets RLOCK
     highestPrice = price;
   currentPrice = price;   // upgrades to WLOCK
  }
}
```

Now imagine two threads with different sessions executing the setPrice() method simultaneously on the same StockQuote object.

If both threads evaluate the expression price > highestPrice before executing the next statement, then each transaction places an RLOCK on the StockQuote object. Then, one of the two threads reaches a subsequent assignment statement. This activity from both threads attempts to upgrade the lock to WLOCK.

However, since the transaction of the other thread also holds an RLOCK, the upgrade attempt will block the thread for a specified amount of time, known as the "lock wait timeout." (This timeout value can be set in the backend profile.)

When the second thread reaches the assignment statement, it also attempts to upgrade the lock to WLOCK. However, the transaction associated with the first thread already possesses an RLOCK on the object. Further, the first thread is waiting for the second thread to release its RLOCK. The result is deadlock, which is detected immediately, and results in the second thread generating a VException with error code 2902, SM_LOCK_DEADLOCK: Deadlock detected.

See also “Error Handling” on page 149.

If the second thread does not release its lock before the first thread times out, then the first thread generates a `VException` with error code 2903, `SM_LOCK_TIMEDOUT:` Lock wait timed out.

One solution to the deadlock problem is to set the `WLOCK` first, so that the transaction does not attempt to upgrade from `RLOCK` to `WLOCK`. In the example above, you can set the `WLOCK` by simply re-ordering the statements. Note that for this solution to work correctly, the object must not already be locked with an `RLOCK`. If that is the case, then the same problem occurs, but more subtly stated.

For example:

```
void setPrice (double price)
{
  currentPrice = price; // sets WLOCK
  if (price > highestPrice)
   highestPrice = price;
}
```

In general, it is not always possible to solve a potential deadlock by merely re-ordering statements. It might be necessary to explicitly set a lock as shown in the example above.

For example:

```
void setPrice (double price)
{
  Handle handle = session.objectToHandle (this);
  handle.upgradeLockTo (Constants.WLOCK, 0);
  if (price > highestPrice)
  highestPrice = price;
  currentPrice = price;
}
```

Note that the `upgradeLockTo()` method is part of Fundamental JVI; therefore it is necessary to first convert the Transparent object to a Fundamental `Handle`. For more information, see Fundamental JVI.

In the Transparent Layer to downgrade the locks on the objects, the method `TransSession.downgradeLocksTo (Object obj[], int lockmode)` is used. If the size of specified object array in this method is zero and the specified lockmode is `NOLOCK`, then this method will release all the locks of the active transaction.

Transitive persistence

Some instances of a class could be persistent (saved in the database) and other instances of a class could be transient (not saved in the database).

In Transparent JVI, an FCO could become persistent in one of the following ways:

- The class of the FCO is categorized as Persistent Always.

  In this case, the object becomes persistent immediately upon creation.
- The object is explicitly made persistent by invoking the `TransSession` methods, `makePersistent()` or `makeRoot()`, passing the FCO as the argument.

If an object is reachable from a persistent object, it could become persistent through transitive persistence (also known as persistence by reachability). The test for reachability occurs when the transaction is committed.

If a Persistent Capable object is referenced by another object that is persistent, then it will also become persistent. For example, suppose that object `L` of a Persistent Capable class or a Persistent Always class contains a reference within it to another object `m1` of a Persistent Capable class. When `L` becomes persistent, `m1` would also become persistent. The situation of transitively determining the set of objects that must become persistent could potentially have a snowball effect, leading to a large chain (or tree) of objects being formed, rooted at the first object becoming persistent. For example, suppose that `m1` contains a reference to other Persistent Capable objects `n11` and `n12`. Consequently, `n11` and `n12` automatically become persistent also. When making an object persistent, the entire sub-tree of references to objects of a persistent class beneath it also becomes persistent.

How to use transitive persistence

To use transitive persistence, define a reference from a persistent object to another object that is Persistent Capable, but has not yet been made persistent.

For example, assume the `Employee` and `Department` classes below are Persistent Capable.

```
class Employee extends Person
{
  Department department;
  double salary;
}

class Department
{
  String name;
  Employee manager;
}
```

Now, the following example creates some Employee and Department objects.

```
Employee  emp = new Employee ();
Department dept = new Department ();
emp.name    = "John Doe";
emp.age     = 30;
emp.salary   = 50000;
emp.department = dept;
dept.name    = "Sales";
dept.manager  = null;
session.makePersistent (emp);
```

This example shows that two objects are created and stored in the database. The Employee object is explicitly made persistent. Thus, it is written to the database when the associated transaction commits. However, the Department object is also written to the database at this time because it is reachable through the department field of the persistent Employee object.

Cache Iterator

JVI Transparent binding makes use of a cache to cache Objects that were accessed previously in a transaction. Caching prevents repeated database access for reading or writing, thereby providing a manifold increase in application execution speed.

The state of a cached object varies based on the way the object was used in the transaction. The state of the object could be new, clean, dirty, deleted or inactive.

- The state of the object is said to be new, if it has not been retrieved from the database but has been made persistent in the current transaction.
- The state of a cached object is said to be dirty if the object has been modified in the current transaction.
- The state of a cached object is said to be deleted if the object has been deleted in the current transaction.
- The state of an object is said to be clean if the object in cache has not been modified or deleted and accessing individual fields of the object will not trigger a database read.

NOTE:- A cached object being clean does not preclude it from being outdated with respect to the object in the database.

- The state of a cached object is said to be inactive if the object has not been read previously and is not in the deleted or dirty states. Accessing any field in the inactive object for the first time triggers a database read. Typically objects transition to the inactive state after a commit operation. However operations such as `commitAndRetain` and `checkPointCommit` which commit and explicitly retain Object state do not cause this transition.

From an application point of view, it can be useful to obtain the list of new, dirty, deleted clean or inactive objects in the cache. Towards this end, TransSession exposes an API.

Enumeration `getCachedObjects (int state)` that returns an Enumeration to iterate over clean, dirty, new, deleted or inactive objects. The objects contained in the Enumeration are based on the state specified matching with the actual state of the cached object. The following states defined in the `class com.versant.fund`.Constants can be used in specifying the state:

```
Constants.CACHED_NEW_OBJECTS
Constants.CACHED_DIRTY_OBJECTS
Constants.CACHED_DELETED_OBJECTS
Constants.CACHED_CLEAN_OBJECTS
Constants.CACHED_INACTIVE_OBJECTS
```

The following code snippet demonstrates how to iterate over some cached objects. In this particular example, the objects that have been modified in the current transaction are being printed.

```
TransSession s = new TransSession("dbName");
Enumeration enum = s.getCachedObjects(Con
stants.CACHED_DIRTY_OBJECTS);
System.out.println (“The following objects have “ +
  “been modified in the current transaction“);
while(enum.hasMoreElements())
 System.out.println (enum.nextElement ());
```

Cached Objects Count

If the application needs to use the count of the cached objects in various states directly, then following API has been provided.

```
int getCachedObjectsCount (int state)
```

This method returns the count of cached, clean, dirty, new, deleted or inactive objects depending on the parameter state.

The parameter state can take the same values as mentioned previously for the `getCachedObjects (int state)` method.

The following code snippet demonstrates how to get the count of cached dirty objects.

In this particular example, the objects that have been modified in the current transaction are being printed.

```
TransSession s = new TransSession("dbName");
int object_count =
s.getCachedObjectsCount(Constants.CACHED_DIRTY_OBJECTS);
System.out.println ("The number of objects modified in the current
transaction are : " + object_count);
```

Second Class Objects

Overview

A Second Class Object (SCO) is an object embedded in an FCO. Therefore, FCOs own the SCO embedded in it.

An SCO has its own identity in Java (i.e. derives from `java.lang.Object`), but does not have its own identity (LOID) in the database.

To explain SCOs, this section describes:

- SCO Java classes
- SCO Tracking

SCO Java classes

Overview

Three basic types of Java classes are SCOs:

1. Classes that map naturally to Versant attribute types such as:
   - Strings
   - Arrays of simple types, such as integers, characters, and floating point numbers.
2. Classes that receive special treatment from the enhancer and an attribute for each of the attributes from its owner. The primary example of this type of SCO is `com.versant.util.DVector.`

3. Classes that are not included in the two previous categories are serialized with the Java serialization mechanism.

These types of SCOs must implement the `java.io.Serializable` or `java.io.Externalizable` interface.

Dual serialization

When an SCO is serialized, it uses a technique called dual serialization. Dual serialization acts like a sieve, separating the FCOs from the byte stream that represents the SCO being serialized. An object that is dual-serialized is written to the database as two attributes:

- An array of bytes that holds the serialized byte stream of the SCO.
- An array of links that contains the references to the persistent objects (i.e. FCOs) contained in the SCO. If the SCO does not contain any references to persistent objects, this array is empty.

How to store SCOs in persistent objects

To store an SCO within a persistent object, simply define the fields of the persistent object with types that are not Persistent Capable or Persistent Always. Examples are given in the `PersistentClassWithSCOs` class below.

For example:

```
class PersistentClassWithSCOs
{
  int[] array;
  java.util.Vector vector;
  Object object;
}
```

Each of the three fields of this class can hold an SCO. The first field, `array`, maps naturally to a Versant attribute type `o_4b[]` also referred to as a "`vstr of o_4b.`"

The second field holds an instance of `java.util.Vector`. This class is not normally enhanced and is categorized as Not Persistent Aware. However, it is serializable, and therefore can be used as an SCO. It is written to the database as a pair of attributes within `PersistentClassWithSCOs`.

One attribute contains the serialized bytes representing the vector. A special, reserved prefix is attached to this attribute name, so that its name is `_vj_serial_vector`. The other attribute contains links to the persistent objects contained within the vector. Its name is simply `vector`.

- If the vector contains many persistent objects, then it probably is more efficient to use the `com.versant.util.DVector` class instead of `java.util.Vector`.

See also "Performance Tuning" on page 197.

The third field can hold any subclass of `java.lang.Object`. Like the `vector` field, the `object` field is dual-serialized. This field can hold any valid FCO or SCO. For example, the `object` field could refer to an integer array, a `PersistentClassWithSCOs` object, or a `java.util.Vector`. All of these cases are correctly handled by the dual serialization mechanism.

SCO tracking

When an application changes a SCO, the FCO that owns the SCO needs to be marked dirty. Marking the FCO dirty ensures that the changed SCO is written to the database with the owning FCO upon a commit. The ability to automatically dirty owning FCO classes is called SCO tracking.

SCO tracking maintains a mapping from an SCO to the owning FCO. When the SCO is modified, the owning FCO is dirtied.

SCO tracking automatically occurs when using the following classes:

- `java.util` package:

  ArrayList, BitSet, EntrySet, HashMap, HashSet, Hashtable, IdentityHashMap, LinkedHashMap, LinkedHashSet, LinkedList, MapEntry, NavigableSet, Properties, SortedMap, SortedSet, Stack, TreeMap, TreeSet, Vector.
- `com.versant.util.DVector`
- Classes categorized as Transparent Dirty Owner

For any other classes of SCOs, the owning FCO must be manually marked dirty with the `TransSession.dirtyObject()` method. This applies also for user defined collection classes implementing the Collection interface.

SCO tracking is not necessary for `java.lang.String` because objects of this class are immutable. This note also applies to other immutable SCOs such as `java.lang.Integer`.

The following describes how you can track SCOs.

How to use SCO tracking

SCO tracking relieves you from manually marking a persistent object as dirty when an SCO is modified. For example, consider the `modifySCO()` method in the following example. This method operates on a `PersistentClassWithSCOs` object. Assume that this object is persistent and has been stored in the database. This method passes an SCO – the `array` field of the `FCO` object – to the `resetArray()` method. The `resetArray()` method modifies the array by setting all of the elements to 0.

The enhancer changes array modifications, such as the statement `array [i] = 0`, so that an owning FCO is marked dirty. In this case, the `PersistentClassWithSCOs` object is the owner of the integer array. Therefore, when the associated transaction commits, the `PersistentClassWithSCOs` object is written to the database, including the changes to the `array` field.

SCO tracking does add runtime overhead to an application.

See also "Performance Tuning" on page 197.

For example:

void resetArray (int[] array)

```
{
  for (int i = 0; i < array.length; i++) {
   array [i] = 0; // marks "fco" as dirty
  }
}
void modifySCO (PersistentClassWithSCOs fco)
{
  resetArray (fco.array);
}
```

Queries 7.0

Introduction

The basic goal for Queries 7.0 is to significantly increase the expressive power of Versant's Query engine, as well as to improve the performance.

To harness the power of the new C APIs designed for query processing, we have introduced two new classes in JVI interface at transparent layer i.e. in `com.versant.trans package` and one class at fundamental layer i.e. in `com.versant.fund package`.

- **Query**

  "Query" is introduced at the transparent layer. It encapsulates one query represented by a query string.

- **QueryResult**

  "QueryResult" is also introduced at the transparent layer. This instance is the result of a query execution. It is an encapsulation of the result set and serves as an iterator.

- **QueryExecutionOptions**

  "QueryExecutionOptions" is introduced at the fundamental layer. This constructor returns a new instance of `QueryExecutionOptions` with all options set to false. The options are encapsulated this way to make sure that only correct options can be passed and no bit operations need to be used to enable several options.

NOTE:- Queries 7.0 is the newer version of queries which users are encouraged to use as Queries 6.0 might be deprecated in future.

Query

Constructor

```
Query(TransSession session, java.lang.String queryString)
Query(TransSession session, java.lang.String database,
java.lang.String
 queryString)
```

The constructor without the database parameter uses the default database of the session. The constructor will compile the query internally and throw an exception if the compilation fails.

Getters

```
java.lang.String getDatabase()
```

Returns the name of the database in which the query will run.

```
java.lang.String getQueryString()
```

Returns the query string.

```
TransSession getSession()
```

Returns the session.

Sole getter methods do not have corresponding setter methods because the query is compiled inside the constructor.

Setter/Getter

```
int getClassLockMode()
```

Returns the class lock mode used by this query.

```
Query setClassLockMode(int mode)
```

Sets the class lock mode for this query.

```
int getInstanceLockMode()
```

Returns the instance lock mode used by this query.

```
Query setInstanceLockMode(int mode)
```

Sets the instance lock mode for this query.

The lock to be placed on returned instances.

By definition, a select operation returns reference to the objects that satisfy your predicate. The default is to not place a lock on the selected objects themselves. If you specify a read, update, or write lock in this parameter, objects are locked by the server if they satisfy the predicate.

This means that during the time of the select operation, you may encounter lock conflicts. It also means that afterwards, you are guaranteed that the objects corresponding to your returned result are consistent and will not change while you are looking at them.

By comparison, if you do not specify a lock in this parameter and then use the returned references to fetch objects in a separate operation, there will be a window of time in which other users can modify or delete your selected objects. Even if you specify fetching options or a non-zero value in levels, a window of time will still exist in which your selected objects may be modified (although it will be much smaller.).

For NOLOCK the fetching and calculation of the closure will use the default lock mode of the session, which is typically a RLOCK. If you want to perform dirty reads then use the session method `setDefaultLock`(NOLOCK) before executing the query.

```
int getFetchSize()
```

Gets the fetch size currently used by this query.

```
Query setFetchSize(int size)
```

Sets the fetch size of this query.

```
QueryExecutionOptions getQueryExecutionOptions()
```

Gets the execution options used by this query.

```
Query setQueryExecutionOptions(QueryExecutionOptions options)
```

Sets the execution options used by this query.

```
Query setCandidateCollection(java.lang.String candidate
               Object [] Objects)
```

Sets the candidate collection for this query.

```
Object [] getCandidateCollection(java.lang.String candidate)
```

Gets the Candidate collection used by this query.

Methods

```
QueryResult execute()
QueryResult execute(QueryExecutionOptions options)
```

The `Query.execute()` method will use the default `QueryExecutionOptions` as provided by the static method

```
QueryExecutionOptions.defaultOptions().
```

`void bind(String parameter, Object value)`

`Query.bind()` can take any single value parameter. With this method, all Date and Time classes from JVI will be translated directly into the correct type needed for the query. Hence users now need not call any methods from class `DateTimeConvert`.

`Query.bind()` will return the Query instance to allow chaining of setter and bind methods.

Clean up

`void close()`

Closes the result set and frees internal resources.

NOTE:- Closing a query is important to prevent memory leaks.

QueryResult

Methods

`boolean isEmpty()`

Determines if this `QueryResult` is empty.

`java.lang.Object next()`

Returns the next object in the result set, or null if no objects are available to return anymore.

`java.lang.Object[] next(int size)`

Returns up to size objects from the result set into an array.

`java.lang.Object[] nextAll()`

Returns the remaining objects in the result set.

`Query getQuery()`

Returns the Query that created this `QueryResult`.

`void close()`

Closes the result set and frees internal resources.

QueryResult encapsulates the query result and serves as an iterator through the result set. It allows the retrieval of individual objects with the `next()` method as well as batch operation with the next(int size) method. The next(int size) method will return up to size objects. The `nextAll()` method will return all remaining objects in the `QueryResult`.

```
java.lang.Object next()
```

where the return value of next() is:

- type PersistenceCapable in case of no projections,
- wrapper type in case of projection on base type fields
- PersistenceCapable in case of reference field
- Object [] in case of more than one column, where each element of the Object [] is of the type as appropriate to the respective column type

QueryExecutionOptions

Constructor

```
QueryExecutionOptions()
```

This constructor returns the default instance of the class `QueryExecutionOptions`.

The fetch operation will place the instanceLockmode on the objects. If the instanceLockmode is NOLOCK then the default lock mode of the session is used.

Setter/Getter

```
boolean isFetchObjects()
QueryExecutionOptions setFetchObjects(boolean fetchObjects)
```

If set, the query will load all objects into the client cache immediately. The fetch operation will place the instanceLockmode on the objects. If the instanceLockmode is NOLOCK then the default lock mode of the session is used.

```
boolean isFlushAll()
QueryExecutionOptions setFlushAll(boolean flushAll)
```

If set, the query will flush all new and dirty objects from the client cache first before executing the query in the server.

```
boolean isFlushNone()
QueryExecutionOptions setFlushNone(boolean flushNone)
```

If set, the query will not flush new and dirty objects from the client cache before executing the query in the server.

```
boolean isPinObjects()
QueryExecutionOptions setPinObjects(boolean pinObjects)
```

If set, objects loaded with fetch all are also pinned in the C Cache.

NOTE:- `FlushAll` and `FlushNone` are mutually exclusive. Setting one of the options to true will automatically set the other option to false if set.

Usage Examples

Simple query

```
Query query = new Query(session_, "select selfoid from t
est.model.Order where date_ > $date");

query.bind("date", new java.sql.Date(105, 3, 10)); // April 10th 2005

QueryResult result = query.execute(new

QueryExecutionOptions().setFetchObjects(true));
Object obj;

while ((obj = result.next()) != null) {
System.out.println("Result : " + obj);
}
query.close();
```

Batch Query

```
Query query = new Query(session_, "select selfoid from
test.model.Order
 where date_ > $date");

query.bind("date", new java.sql.Date(105, 3, 10)); // April 10th 2005

QueryResult result = query.execute(new queryExecutionOptions().setF
etchObjects(true));

int batchSize = 50;

while (true) {
Object[] objs = result.next (batchSize);
displayBatch(objs);
if (objs.length < batchSize) {
break;
}
}
```

```
result.close();

query.close();

Collection Query:

Query query = new Query(session_, "select selfoid from
 test.model.Order where date_ > $date");

query.bind("date", new java.sql.Date(105, 3, 10)); // April 10th 2005

QueryResult result = query.execute();

Object[] obj = result.nextAll();
processOrders(Arrays.asList(obj));
query.close();
```

Queries 6.0

Introduction

Queries allow you to find persistent objects that match a specified condition. (example: "`salary < 100000`".)

Queries are evaluated in the back-end (server), and the results are passed to the front-end (client).

This section describes:

- Predicate objects
- VQL
- VEnumeration
- Indexes
- Roots

Predicate objects

A predicate object can be passed to the `TransSession.select()` method.

Predicate objects are part of Fundamental JVI.

For more information refer to “Predicate objects” on page 56, in "Chapter 2 - Fundamental JVI".

You can also use the `TransSession.select()` method to select all instances of a class by passing in null for the predicate object.

How to use predicate objects

Using the `TransSession.select()` method is similar to using the `ClassHandle.select()` method in Fundamental JVI. Both methods take predicate objects to specify which objects should match the query. If a `null` predicate is given to the `select()` method, then all instances of that class (or all instances of that class and its subclasses if the option `Constants.SELECT_DEEP` is used) result from the query.

For example:

```
Enumeration e = session.select (Person.class, null);
while ( e.hasMoreElements() ) {
  Person person = (Person) e.nextElement ();
  System.out.println ("Found " + person);
}
```

To build a predicate, use the Fundamental JVI `Attr` objects. These objects have methods, such as `eq` (for "equals") and `ne` (for "not equals") that construct predicate objects.

For example:

```
AttrString attr_name = session.newAttrString ("name");
String[] names = { "Bob", "Joe", "John" };

for (int i = 0; i < names.length; i++) {
  Enumeration e = session.select ( Person.class,
                  attr_name.eq (names [i]),
                  "db@server",
                  Constants.SELECT_DEEP,
                  0,
                  Constants.IRLOCK,
                  Constants.NOLOCK );

  while ( e.hasMoreElements() ) {
   Person person = (Person) e.nextElement ();
   System.out.println ("Found " + person +
             " with name " + names [i]);
  }
}
```

Note that in the example above, the `select` method is given several arguments. These arguments specify various query options such as class and instance lock modes, database name, and flushing behavior.

For more information, on these options, please refer to Section on `TransSession.select` in the *JVI javadoc*. You can find more information on using `Attr` and `Predicate` objects in the chapter *"Fundamental JVI"* in this Manual.

VQL Queries

VQL (Versant Query Language) specifies queries as strings instead of predicate objects. Internally, VQL queries convert to predicate objects.

Transparent JVI expresses VQL queries by creating a `VQLQuery` object. Consequently, the constructor accepts a query string argument.

Some values, such as arrays and strings, are difficult or impossible to express in VQL. To solve this problem, use substitution parameters. A substitution parameter is written as `"$1"`, `"$2"`, and so on, in the query string. You can then substitute the values with the `bind()` method on the `VQLQuery` object.

To find the objects that satisfy your query string, invoke the `execute()` method on the `VQLQuery` object.

How to use VQL queries

Using a VQL query requires four steps:

1. Create a `com.versant.trans.VQLQuery` object, passing the query string to the constructor. The query string should be of the form "`select selfoid from ...`".
2. Bind arguments to the substitution parameters (`$1, $2,` and so on) in the query string. If the query does not have substitution parameters, then skip this step.
3. Invoke the `VQLQuery.execute()` method. This method returns an object that implements the `com.versant.trans.VEnumeration` interface.
4. Use the `Enumeration.hasMoreElements()` and `Enumeration.nextElement()` methods to obtain all of the objects that satisfy the query.

The following example illustrates the use of a simple VQL query.

```
VQLQuery vql = new VQLQuery (session, "select selfoid from Person");
Enumeration e = vql.execute ();
while ( e.hasMoreElements() ) {
  Person person = (Person) e.nextElement ();
  System.out.println ("Found " + person);
}
```

When a `VQLQuery` object is created, it can be re-used many times as shown in the following example.

```
VQLQuery vql = new VQLQuery
  (session, "select selfoid from Person where name = $1");
String[] names = { "Bob", "Joe", "John" };

for (int i = 0; i < names.length; i++) {
  vql.bind (names [i]);
  Enumeration e = vql.execute ();
  while ( e.hasMoreElements() ) {
   Person person = (Person) e.nextElement ();
   System.out.println ("Found " + person +
             " with name " + names [i]);
 }
```

The `VQLQuery.execute()` method can also accept more advanced options, such as lock modes for class objects and instances. By default, the query does not place a lock on instances. To ensure that the results of the query remain consistent, place an `RLOCK` on the instances. Alternatively, if the objects are to be modified, `ULOCK` (update lock) or `WLOCK` can be used.

For example:

```
VQLQuery vql = new VQLQuery (session, "select selfoid from Person");
Enumeration e = vql.execute (0, 0, Constants.IRLOCK, Cons
tants.RLOCK);
while ( e.hasMoreElements() ) {
  Person person = (Person) e.nextElement ();
  System.out.println ("Found " + person);
}
```

How to query for a Boolean Value using a VQLQuery

A field of type boolean in user's persistent classes is mapped to a o_u1b type (one byte unsigned integer) in the Versant database. The following examples show the way in which a boolean field can be queried:

Example 1. Executing VQLQuery using the bind method

```
VQLQuery q = new VQLQuery (session,
  "select selfoid from Person where married = $1");
q.bind ( new Boolean (false) );
VEnumeration e = q.execute ();
```

Example 2. Executing VQLQuery without the bind method.

As boolean are stored as one byte unsigned integers, 1 is treated as true and 0 is treated as false. The query looks as follows:

```
VQLQuery q = new VQLQuery (session,
  "select selfoid from Person where married = 1" );
e = q.execute();
```

How to bind a NULL value to java.utilDate / java.sql.TimeStamp using VQLQuery

A field of type `java.util.Date` or `java.sql.Timestamp` in user's persistent classes is mapped to a `o_timestamp` type (8 byte integer) in the Versant database.

The following examples show the way in which a `NULL` date field can be queried:

```
VQLQuery q = new VQLQuery (session,
  "select selfoid from Person where birthday = $1");
q.bind ( new Long(DateTimeConvert.NULL_O_TIMESTAMP) );
VEnumeration e = q.execute ();
```

For conversion between Versant time and date formats (`o_date`, `o_time`, and `o_timestamp`) to Java time and date objects (`java.util.Date,java.sql.Date, java.sql.Time`, and `java.sql.Timestamp`), APIs of class "`com.versant.util.DateTimeConvert`" is used.

VEnumeration

The `VQLQuery.execute()` and `TransSession.select()` methods return a `VEnumeration` object.

A `VEnumeration` object allows you to iterate through all of the objects that result from a query.

Since the `VEnumeration` interface extends the `java.util.Enumeration` interface, it can be used in the same way as an `Enumeration`. It also contains the `nextHandle()` method that returns the next element in the enumeration already cast to a `Handle`.

Using the query results in the VEnumeration

When a VQL query is used to perform a query, the set of objects that satisfy the query are returned in the form of a `VEnumeration` instance. Most applications iterate through the enumeration by calling the method `nextElement` on the Enumeration object. The `nextElement` call is then typically followed by a access to a field in the object. This field access can result in the object being read from the database if the object is not present in the Java Cache.

Queries can result in a lot of instances being selected from the database. For such queries the process of iterating through all of the instances in the `VEnumeration`, reading the instance from the database if necessary and using the instance in the application can become a huge performance bottleneck. Performance can be boosted immensely if we can reduce the number of network calls made to the database server to fetch objects into the client cache.

Towards the end the functionality of the `nextBatch` API in `VEnumeration` has been modified. This API will group read a subset of objects and return this subset of the query result set. The size of the subset that is returned and the lock mode used to group read the objects can be specified by the user.

For more information, about the `nextBatch` APIs, please refer to the javadoc for the class `VEnumeration`.

The following example demonstrates the usage of the `nextBatch` API:

```
VQLQuery query = new VQLQuery (session, “select * from Person”);
VEnumeration venum = query.execute();
while (venum.hasMoreElements ()) {
 // The object array contains the Person objects that satisfied the
 // query.These Person objects have been materialized into the java
 // cache. Accessing the fields of any of these Person instances will
not // cause a implicit fetch from the database.
  Object[] = venum.nextBatch (batchSize, options, lockmode);
  // use the objects in the array.
}
```

NOTE:- In case of cursor queries (`selectWithCursor` or `executeWithCursor`), calls to API’s `nextElement` and `nextBatch` should not co-exist on the same cursor object to fetch all instances satisfied by the cursor query. If used it can result in unpredictable behavior. Only use either `nextElement` or `nextBatch` in conjunction with / without `hasMoreElements`.

The following example uses cursor-based query to find all instances of `Person`.

```
VQLQuery query = new VQLQuery (session, “select * from Person”);
VEnumeration venum = query.executeWithCursor();

while (venum.hasMoreElements ()) {
 Object person = venum.nextElement ( );
}
```

Or,

```
while (venum.hasMoreElements ( ) ) {
 Object[] array = venum.nextBatch (10);
}
```

Indexes

You can use an index to make a query execute more quickly. However, with an index, you risk increasing overhead when you update objects.

Indexes are created with Fundamental JVI.

For more information refer to “Indexes” on page 58, in "Chapter 2 - Fundamental JVI".

Roots

You can query individual objects by name. These named objects are called roots. A root is an object that has a unique string persistently attached to it.

To name an object, call the `TransSession.makeRoot()` method. Later, if you want to find the object, you can pass the name to the `TransSession.findRoot()` method.

An object can have more than one name. To remove a name, you can use the `TransSession.deleteRoot()` method. This method only deletes the name; the object is not deleted.

The following describes how you can define, find and delete roots.

How to define roots

To name an object, call the `TransSession.makeRoot()` method. You can find the object again later using the `TransSession.findRoot()` method. Note that a persistent object can easily have more than one name.

The following example associates the name "customer list" with a persistent object, `customer_list`. This object must be a First Class Object. If the root name is already in use, an exception is thrown.

See also “Error Handling” on page 149.

For example:

```
try {
  session.makeRoot ("customer list", customer_list);
} catch (VException exception) {
  if (exception.getErrno() == Constants.EVJ_ROOTNAME_ALREADY_FOUND)
   System.out.println ("Root name already exists");
  else
   throw exception;
}
```

How to find roots

When a root has been defined, you can find the associated object with the `TransSession.findRoot()` method. The following example locates the customer list using the "customer list" root name. If that name is not associated with any object, an exception is thrown.

For example:

```
try {
  customer_list = (CustomerList) session.findRoot ("customer list");
} catch (VException exception) {
  if (exception.getErrno() == Constants.EVJ_ROOTNAME_DOES_NOT_EXIST)
   customer_list = null;
  else
   throw exception;
}
```

How to delete roots

To delete an associated root name, use the `TransSession.deleteRoot()` method. The following example deletes the association between the customer list, so that a new list can be associated with that name. If no object exists with that name, an exception is thrown.

Note that the `deleteRoot()` method deletes only the associated name; it does not delete the object itself. To delete the object, use the `TransSession.deleteObject()` method.

NOTE:-

- If the server profile parameter `commit_delete` is `OFF`, this function will send the delete request to the source database and the object is deleted immediately.
- If `commit_delete` is `ON`, this function will acquire a Write Lock on the object and set its status as "`Marked for deletion`". The object will be physically deleted at commit.

  If the transaction commits, the objects are physically deleted. If a rollback occurs, these objects are un-marked and their status restored.
- Queries run on the database will not include objects marked for deletion in the result sets.

For example:

```
try {
  session.deleteRoot ("customer list");
} catch (VException exception) {
  // Ignore error if root name does not exist
  int err = exception.getErrno ();
  if (err != Constants.EVJ_ROOTNAME_DOES_NOT_EXIST) {
   throw exception;
}
session.makeRoot ("customer list", customer_list);
```

Queries using Virtual Attribute Template (VAT)

Overview

To enhance query capability, the concept of Virtual Attributes has been introduced. A Virtual Attribute is a fictitious attribute that is not saved in the database but is computed on demand. Virtual Attributes can be indexed. These indices are maintained during normal object operations (insert, delete and update) and are used during query processing.

For more information, about Virtual Attributes and Virtual Attribute Templates (VAT), please refer to Section “Advanced Queries” in Chapter “Query Processing With VQL 6.0” in the *Versant Database Fundamentals Manual.*

How to use Virtual Attributes in VQL Queries

The example in the VQL query section above can be modified to use VAT as follows:

```
VQLQuery vql = new VQLQuery
  (session, "select selfoid from Person where `/national De_de utf8
                                              name` = $1");
String[] names = { "Beßle-Schmidt", "Joe", "{Jürgen Uwe}" };
for (int i = 0; i < names.length; i++) {
  vql.bind (names [i]);
  Enumeration e = vql.execute ();
  while ( e.hasMoreElements() ) {
   Person person = (Person) e.nextElement ();
   System.out.println ("Found " + person +
             " with name " + names [i]);
 }
}
```

For more information, about using predicate objects and VAT, please refer to Section “Queries using Virtual Attribute Template (VAT)” on page 67 in "Chapter 2 - Fundamental JVI".

Error Handling

Methods in Transparent JVI do not return error codes. Instead, a runtime exception `VException` is thrown if an error occurs.

A `VException` can be generated when a method is not explicitly called. Accessing an FCO can result in the object being fetched from the database; this process can generate an error.

`VException` is part of Fundamental JVI.

For more information refer to “Error Handling” on page 73, in "Chapter 2 - Fundamental JVI".

The following describes how to catch exceptions when you access an array of persistent objects.

How to catch exceptions when accessing an array of persistent objects

In the following example, `courses` is an array of persistent `Course` objects. Accessing this array can generate exceptions for each element of the array. The following example displays the individual exceptions for each element of the array.

```
try {
  int length = student.courses.length;
  System.out.println ("You are taking " + length + " courses");
} catch (VException vex) {
  if (vex.getErrno() ! = Constants.EVJ_EXCEPTIONS_ACCESSING_ARRAY)
   throw vex;

  Vector vector = vex.getArgs ();
  for (int i = 0; i < vector.size (); i++) {
   Exception exception = (Exception) vector.elementAt (i);
   if (exception != null)
     System.out.println ("Error accessing element " + i +
               ": " + exception);
  }
}
```

Collections

Overview

Transparent JVI supports a variety of collection classes. These collection classes are persistent versions of collection classes typically available in Java.

To explain collections, this section describes:

- Java arrays
- `com.versant.util` package
- ODMG collections

Java arrays

Standard Java arrays can be used in persistent objects.

Java arrays are SCOs. They never have a separate identity (LOID) in the database.

com.versant.util package

The `com.versant.util` package supports persistent versions of the standard Java collections, Vector `and` Hashtable`.`

`VVector`

- `VVector` is the persistent version of `java.util.Vector` and supports the same methods.
- `VVector` is an FCO with its own LOID when stored in the database.
- `VVector` can contain both FCOs and serializable SCOs as elements.

`DVector`

`DVector` is similar to `VVector`, except `DVector` objects are embedded SCOs instead of FCOs.

`LargeVector`

- `LargeVector` is similar to `VVector`. However, `LargeVector` has better support for handling a vector with large numbers of elements.
- `LargeVector` can contain only FCOs.

`VHashtable`

- `VHashtable` is the persistent version of `java.util.Hashtable` and supports the same methods.
- `VHashtable` is an FCO.
- Keys and values can be either FCOs or serializable SCOs.

ODMG collections

The collections in the ODMG packages, `com.versant.odmg` and `com.versant.odmg3`, can also be used in Transparent JVI. All the classes in the ODMG packages are stored as FCOs. These collections include:

```
ListOfObject
SetOfObject
BagOfObject
MapOfObject
```

For more information refer to "ODMG 2.1 Collections" on page 264, in "Chapter 4 - ODMG JVI".

Security

Versant databases are made secure from unauthorized access in two ways:

- User Authentication
- Authorization

User Authentication

Versant authenticates a user attempting to connect to a database by using information stored in the database. A password based authentication scheme is used to authenticate the client connecting to the database server. Before a client can connect to the server, the user name and password provided by the client is compared with the encrypted password stored in the database. The connection is established when there is a match.

If the password entry corresponding to the user is empty or `NULL`, then Versant will authenticate that user for backward compatibility cases.

The user name and password may have to be provided as session properties when a session is instantiated. If the `userName` and `userPassword` are not specified in the Properties object during session instantiation, then the OS user name and a `NULL` password are passed to the server as the database user name. If the OS user is not in the database user list, the connection to the database is not established and session instantiation will fail. Even if the OS user is in the user list, the connect request to the server will fail if no password is specified at the time of session creation and a password was set for the user in the database previously.

As an example consider the following code snippet:

```
Properties p = new Properties();
p.put(“database”, “mydb”);
 // OS user name may or may not be John, but in any case would
 // like to connect as John.
p.put(“userName”, “John”);
// Password is set for John in the database user access list.
// So we need to provide the password for user John.
 p.put(“userPassword”, “xxx123”);
TransSession session = new TransSession(p);
```

Since database creation utilities like `createdb` and `createreplica` currently do not allow the password to be supplied during database creation, the DBA or the Database Administrator is the OS user who invoked the utility. The DBA has a `NULL` password and password authentication is bypassed for the DBA. Since Versant relies on the OS to authenticate the DBA, the OS user name and the DBA name should match for the client to connect to the database as a DBA. Password authentication for DBA will be addressed in future releases of Versant.

Once the connection has been established no further authentication is done for threads joining the existing session. This means that after a session has been established, the user name and password for any thread joining the session is not checked. Users have to implement their own authentication mechanism for threads joining a session that has already been established The user name and password provided are cached on a per thread basis. The `FundSession` API `setUserLogin` can be used to set the user name and password for any thread. The OS user name and a `NULL` password is used as the default user name and password for a thread which does not have these two parameters set.

User authentication is also done when an existing session on a database wants to connect to another database. The `userName` and `userPassword` cached for the thread is used to authenticate the user before the connection to the other database is established. The example below shows how the current user name and password can be changed.

```
Properties p = new Properties();
p.put(“database”, “db1”);
p.put(“userName”, “John”);
p.put(“userPassword”, “xxx123”);
// begin session with user name John and password xxx123
```

```
  TransSession session = new TransSession(p);
 // need to connect to db2 as John with password xxx123. The
// user name and password has already been set for this
// thread. Nothing more other than the connect needs to be
// done.
session.connectDatabase(“db2”, Constants.READ_WRITE_ACCESS);
// need to connect to ‘db3’ as Jane. Change the user name and
// password that is cached within this thread.
session.setUserLogin(“Jane”, “yy123”);
// connect to database db3 with user name Jane and pass yy123.
session.connectDatabase(“db3”, Constants.READ_WRITE_ACCESS);
```

User Authorization

User Authorization deals with a user’s access privileges. Any valid database user can be provided with both read and write access or only read access.

For any Versant database, a user can be thought of as having one or more of the following roles:

1. The Database User or dbuser
2. The Database System Administrator or DBSA
3. The Database Administrator or DBA

The database users are the list of Versant maintained logical users who have access to a particular database. The database user need not have a valid operating system user account. Based on the DBA’s discretion a user can be authorized for `READ` only access or for `READ/WRITE` access.

The DBA for a database is the user who created the database. The database user list for a particular database can be created and maintained by the DBA of that database. The DBA is required to have a valid operating system user account. The DBA is automatically assigned `READ/WRITE` access.

The DBSA or the Database System Administrator is the user who installs Versant on each machine in a network.

How to add a user to the database user list

A user can be added to the database user list using the `dbuser` utility. Programmatically the user can also be added to the database user list using the `DBUtility` APIs.

For more information, on how to add a user using the dbuser utility, please refer to the Chapter “Database Utilities” in the Versant Database Administration Manual.

The DBUtility class provides APIs to add a user to the database, change the user password, delete a user, obtain the current database user list and to provide public access to any database.

For more information, please refer to the DBUtility class in the *JVI javadoc.*

Secure Sessions

In some internet uses of Java, it is commonplace for some threads in the Java Virtual Machine to execute downloaded code automatically from the internet. For example, this download could be a side effect of visiting a web page.

From a security point of view, such code is termed untrusted and should be regarded as a potential adversary. Java's SecurityManager protects most operating system resources from untrusted code. However, the SecurityManager does not protect the native methods that give Java access to a Versant database.

To protect Versant databases, the NewSessionCapability class was created. Both Fundamental JVI and Transparent JVI enforce the rule that only one instance of NewSessionCapability can be created inside any Java Virtual Machine.

A single instance of NewSessionCapability must be created by trusted code, and a reference to it should only be given to other trusted code that can create new sessions.

If you want to create secure sessions, you must instantiate an instance of NewSessionCapability before you can begin a session with a Versant database. NewSessionCapability implements the Capability interface as follows:

```
Capability capability = new NewSessionCapability ();
```

If there is any untrusted Java code in your Java interpreter, the NewSessionCapability instance must be instantiated first with trusted code before any untrusted code runs.

Untrusted code should not be allowed to access the NewSessionCapability object. Since only one instance of NewSessionCapability can be created in each Java interpreter, any subsequent attempt to create an instance results in an exception.

After instantiation, every session that is created would need to use the single NewSessionCapability instance.

To explain how to set session security, this section describes how to create a secure session.

How to create a secure session

You can create secure sessions with a Versant database using an instance of `Capability`. The `Session` constructor accepts a `Capability` instance as a parameter.

The public class `NewSessionCapability` implements the `Capability` interface and its constructor enforces that only one instance can be created for each Java class loader.

A single instance of `Capability` must be created by trusted code. A reference to it should only be given to other trusted code that can create new sessions. When an instance of `Capability` is created, all other `Session` constructors need to use it.

For example:

```
// The Capability object is shared across threads
Capability cap = new NewSessionCapability ();

// ThreadA:
Properties prop = new Properties ();
prop.put ("database", "mydb@myserver");
TransSession session = new TransSession (prop, cap);

// ThreadB:
Properties prop = new Properties ();
prop.put ("database", "mydb@myserver");
TransSession session = new TransSession (prop, cap);
```

Alternately, an "insecure" mode can be used when instantiating sessions. A `Session` constructor that does not accept a `Capability` parameter can be used to instantiate such sessions.

For example:

```
// Thread1:
TransSession session = new TransSession ("mydb@myserver");
// Thread2:
TransSession session = new TransSession ("mydb@myserver");
```

When you create sessions without using a `Capability` argument, instantiation of a new `NewSessionCapability` leads to a `VException` being thrown.

Therefore, when you use `Session` constructors that do not use a `Capability` argument, you work in an "insecure" mode where any user thread can start a session on the database.

ADVANCED CONCEPTS

Schema Objects

A schema object is the database representation of a persistent class; this object contains (among other things) the list of all attributes and attribute types that make up a persistent object. Every persistent object has an associated schema object. When you run the `db2tty` utility (with only the database name option), what you see is a list of schema objects. Persistent objects can be stored in a database only after the corresponding schema has been defined in the database.

Schema objects can be defined either automatically or manually.

Automatic Schema Definition

The JVI Transparent binding provides automatic, runtime schema-definition in the database. Whenever an object is made persistent, the schema for that object will be defined if it does not already exist. While this is a useful feature that extends the transparency of the JVI interface, it must be used carefully. The following database features interact with this automatic schema definition:

- **Locking -** When first defined, the schema object for a class is given a `WLOCK`. This prevents any simultaneous transactions from creating instances of this class until the transaction that defined the class is committed. If the defining transaction does not commit quickly enough, then other transactions will receive "lock wait timeout" errors.
- **Isolation -** If the locking problem is explicitly circumvented by downgrading the lock on a class object, then other transactions will not be isolated from failure of the defining transaction. Consider the following scenario: transaction A defines some class Person, downgrades the lock on the Person schema object, but does not yet commit. Transaction B meanwhile creates some person instances (this is possible only because the lock was downgraded or dropped) and commits. Now, if transaction A fails (rolls back), then the Person class and all of its instances (in particular, those committed in transaction B) will be lost!

Manual Schema Definition

Due to the potential problems described above, a recommended practice when using the Transparent binding in a multi-session environment (see "Threads and Multiple Sessions" in this chapter) is to define all necessary classes in a separate transaction. You can do this in two ways:

- Your application can call the `TransSession.defineClass()` method. This method has a Java class object parameter. Your application should invoke this method in a separate, initial transaction. This transaction should commit before instances of the class are created.
- You can run the `com.versant.DefineClass` command-line utility. This Java application makes it easy to define classes independently of your application. With the DefineClass utility, you can define new classes, redefine classes that have changed, or perform controlled schema evolution when attributes are added or dropped.

DefineClass Usage

```
java com.versant.DefineClass [-evolve] [-redefine] [-noprompt] \
  –db <database name> <class or file 1> <class or file 2> ...
```

The DefineClass application takes a list of class and/or file names. If a class name is specified, it must be fully-qualified (with full package specifier). For example, the following command defines schema objects in the “`mydb`” database for the Employee and Department classes (located in the `com.mycorp package`):

```
java com.versant.DefineClass –db mydb \
com.mycorp.Employee com.mycorp.Department
```

If a file name is specified, it can be one of the following:

- A class file.
- A directory. In this case the directory will be recursively searched for class files.
- A jar file. In this case the jar file will be searched for class files.

For each persistent class encountered by the DefineClass application (including those contained in directories or jar files that are searched), a corresponding schema object will be created in the database.

The `-evolve` and `-redefine` options are explained in the *“Schema Evolution”* section below.

Prompting

Before the DefineClass application defines or changes any schema objects in the database, it will prompt the user for verification. This prompt gives the user a chance to abort the operation before permanent changes are made. If this prompt is not desired, the `-noprompt` option can be used to disable it. This option should be used carefully, since it removes the user's last opportunity to abort the changes to the database.

Schema Evolution

A schema object and the corresponding Java class must be compatible. This compatibility is checked at runtime when an instance of a persistent object is accessed. An incompatibility occurs when the Java class changes (after the schema object has been defined), or when another application changes the database schema (while the Java class remains unchanged). There are three different types of schema incompatibilities:

- A Java class could contain new attributes that are not present in the database schema.
- A Java class could lack attributes that are present in the database schema.
- A Java class could have an attribute of a different Java type compared to the attribute type defined in the database.

Schema Compatibility Options

To control what happens when there is a schema mismatch at runtime, use the `TransSession.setSchemaOption()` method. This method allows you to choose between one of three options for handling schema mismatches:

SCHEMA_ADD_DROP_ATTRIBUTES

If the Java class contains a different number of attributes than the database class, then modify the database class definition so that the database class has the same attributes as the Java class.

If attributes are added, existing database instances will be initialized with null values for each of the newly added attributes.

Dropping attributes means that values for those attributes are deleted from the database.

SCHEMA_FORCE_DROP_DATABASE

If there is any mismatch between the Java class and the database class, drop the database class entirely. This will lead to deleting all instances of the class from the database. A new class object will be defined in the database to match the Java class.

SCHEMA_THROW_EXCEPTION_ALWAYS

If there is any mismatch between the Java class and the database class, raise an exception that is an instance of `VSchemaException`.

This is the default option if you do not call `TransSession.setSchemaOption()`.

When you invoke `TransSession.setSchemaOption()`, your choice of schema handling options affects the current session only.

Each time you begin a new session, the default will be `SCHEMA_THROW_EXCEPTION_ALWAYS`. If you are using the default option `SCHEMA_THROW_EXCEPTION_ALWAYS` and are concerned about schema changes in a database, you might want to execute all database operations in a try-catch block so that you can catch any exceptions. The application could be written so that if a `VSchemaException` was detected by the catch block, the session would be rolled back, and the database operations redone after setting the schema evolution option to either `SCHEMA_ADD_DROP_ATTRIBUTES` or `SCHEMA_FORCE_DROP_DATABASE`.

VSchemaException Class

A `VSchemaException` exception is generated when there is a schema mismatch, and you have not chosen either of the `SCHEMA_ADD_DROP_ATTRIBUTES` or `SCHEMA_FORCE_DROP_DATABASE` options.

`VSchemaException` extends the standard JVI exception `VException`. The error number associated with the `VSchemaException` can be obtained with the `VException.getErrno()` method. This error number will be one of the following, depending on the type of schema mismatch:

- `Constants.EVJ_SCHEMA_DB_ATTR_MISSING`
- `Constants.EVJ_SCHEMA_JAVA_ATTR_MISSING`
- `Constants.EVJ_SCHEMA_JAVA_DB_TYPE_MISMATCH`

The `VSchemaException` object contains several attributes that are set to allow a user program to determine what caused the schema exception to be thrown:

attrName

A Java string that is the name of the attribute in the class for which the exception is thrown. To get this value, use the `VSchemaException.getAttrName()` method.

className

The Java class where the schema incompatibility occurs. To get this value, use the `VSchemaException.getClassName()` method.

dbType

A Java string that represents the Java type of the attribute when the schema was defined in the database. To get this value, use the `VSchemaException.getDbType()` method. The dbType value is not set if category is `ATTRIBUTE_MISSING_IN_DB`.

javaType

A Java string that represents the current Java type of the attribute in the Java class. To get this value, use the `VSchemaException.getJavaType()` method. The `javaType` value is not set if category is `ATTRIBUTE_MISSING_IN_JAVA`.

Manual Schema Evolution

You can use the `DefineClass` utility to manually evolve the schemas for classes that have changed. This application supports two options, `-evolve` and `-redefine`, which correspond directly to the `SCHEMA_ADD_DROP_ATTRIBUTES` and `SCHEMA_FORCE_DROP_DATABASE` options described above.

Adding and Removing Attributes

Use the `-evolve` option to evolve the schema when attributes have been added to or removed from existing classes in the database. This option will modify the schema objects, but existing instances of this class will remain in the database.

If a new attribute is added to a schema object, any existing instances of this class will see a “default” value for this attribute when read. This default value is zero for numerical types and null for references.

As an example, consider the following simple persistent-capable class:

```
class Person
{
String name;
  int age;
}
```

Assume there are already instances of the Person class in a database when a new attribute is added:

```
class Person
{ String name;
  int age;
String gender;
}
```

To evolve the schema in the database "`mydb`" to match the new Java class, you can use the `DefineClass` application as follows:

```
java com.versant.DefineClass –db mydb -evolve Person
```

This results in the following output:

```
DefineClass utility, JVI 7.0.1. Copyright (c) 2005 Versant
Corporation
Person ................................. gets Attribute gender
OK to apply Changes ? (Enter y or yes)
```

If you choose to apply these changes (or if you also use the `-noprompt` option), then the Person schema object will evolve gain an additional attribute, "gender." The existing Person instances in the database will evolve as well.

NOTE:- Transparent JVI (`com.versant.Enhance`, `com.versant.DefineClass`, `com.versant.Launch`) has its own mechanism to order attributes in database whereas using Fundamental JVI, the user can influence order in database by using `AttrBuilder[], dropAttr(), renameAttr(), insertAttrsAt()`.

The API's, `copyObject()` and `migrateObjects()` work only if schema in source and target database are identical. They will fail if order of attributes are different despite the fact that attributes and their domains are equal.

So if the database schema is modified using Fundamental JVI APIs, then `copyObject()` and `migrateObjects()` will throw error `E6063:SCH_MIGR_NO_VALID_OBJS`.

Redefining Schema Objects

When a schema mismatch exists, it is possible to completely redefine the schema, instead of evolving it. When the schema is redefined, the old schema is dropped and a new schema created. All of the old instances are deleted as well, so use this option with caution!

The previous Person class can be redefined in the database "`mydb`" with the following command:

```
java com.versant.DefineClass -db mydb -redefine Person
```

This results in the following output:

```
DefineClass utility, JVI 7.0.1. Copyright (c) 2005 Versant
```

```
Corporation
Person ................................. Redefine
OK to apply Changes ? (Enter y or yes)
```

If you choose to redefine the schema, then a new Person schema object will be created in place of the old one. All of the existing Person instances will be removed from the database.

Threads and Multiple Sessions

Overview

A single Java application can consist of several threads operating on one or more sessions.

Each session has its own transaction, locks and object cache.

To explain threads and multiple sessions, this section describes:

- Setting a session
- Naming a session
- Using a programming model for multiple threads

Set a session

Before a thread can access a session or any objects associated with that session, the thread must join the session by invoking `setSession()`. After using the session, the thread must leave the session by invoking `leaveSession()`.

When a thread instantiates a session, it automatically joins that session unless the `DONT_JOIN` option was set in the properties.

Each thread can be in only one session at a time. When a thread joins a session, it automatically leaves any previously joined session.

Name a session

Every session attached to a database must have a unique session name.

A unique session name applies to both sessions in the same process and in different processes.

If a session name is not explicitly given, a unique name is generated automatically.

Use a programming model for multiple threads

There are two basic programming models when using multiple threads and sessions:

- Multiple Threads, Multiple Sessions (MTMS)
- Multiple Threads, Single Session (MT1S)

Multiple Threads, Multiple Sessions (MTMS)

In this model, each thread that accesses the database has its own session.

MTMS is the simpler model because the database kernel handles all synchronization issues.

In MTMS, a thread attached to a session cannot access persistent objects associated with another session. Objects associated with a session cannot refer to objects associated with another session.

How to use the MTMS model

The following is a complete example that uses the MTMS model. The program has three threads, and each thread has its own session. None of the threads accesses more than one session, and no session is accessed by more than one thread.

One of the purposes of this example is to illustrate that by using the MTMS model, the application does not need to do any explicit synchronization. Instead, the database kernel performs all of the synchronization. Since each thread is associated with its own transaction, each thread can independently lock the objects that are being accessed.

The example is a very simple stock market simulation. It consists of three threads: a main thread, a reader thread and a writer thread.

The main thread simply prepares the database by populating it with `StockQuote` objects for the various stock symbols, spawns the reader and writer threads, and then waits for these threads to finish.

The reader thread periodically prints out a "Stock Market Report" showing the various stock symbols and the current price for that stock. The writer thread constantly updates the various `StockQuote` objects by randomly changing their prices.

Note that the reader thread uses a VQL query to get the `StockQuote` objects to be displayed in the "Market Report". When the query is executed, it places an `RLOCK` on each of the matching objects (in this case, all `StockQuote` objects). Until this transaction commits and releases the locks, no other transaction (such as the writer thread) can place a WLOCK on these objects. Thus, within this transaction, the reader thread sees a consistent "snapshot" of the StockQuote objects.

In a real application, it might not be necessary or desirable to have this snapshot, since this increases lock contention. In this case, the query can place NOLOCK on the objects, which is the default locking option for VQLQuery.

The writer thread locates individual StockQuote objects using a VQL query to match a randomly chosen stock symbol. Here, the query places a WLOCK on the matching object. In this example, it is not strictly necessary. However, if there were multiple writer threads working simultaneously, deadlock could result if the query did not place an initial WLOCK on the object. By default, the transaction would upgrade an RLOCK to a WLOCK. If another thread performed the same action on the same object, deadlock could ensue.

See also “Implicit locking” on page 121.

For example:

```
import com.versant.fund.*;
import com.versant.trans.*;
import java.util.*;

public class StockMarket
{
 public static void main (String[] args)
 {
  if (args.length != 1) {
   System.out.println ("Usage: java StockMarket <database>");
   System.exit (1);
  }
  database = args [0];

  TransSession session = new TransSession (database);
  for (int i = 0; i < symbols.length; i++) {
   StockQuote quote = new StockQuote ();
   quote.symbol = symbols [i];
   quote.price = Math.abs(random.nextInt()) % 10000;
   session.makePersistent (quote);
  }
  session.endSession ();

  new StockMarketReader().start();
```

```
   new StockMarketWriter().start();
  }

  static String randomSymbol ()
  {
   int index = Math.abs(random.nextInt()) % symbols.length;
   return symbols [index];
  }

  static String database;
  static boolean done = false;
  static Random random = new Random ();
  static String[] symbols = { "vsnt", "sunw", "ibm", "msft", "orcl" };
 }

 class StockMarketReader extends Thread
 {
  public void run ()
  {
   TransSession session = new TransSession (StockMarket.database);
   VQLQuery vql = new VQLQuery
    (session, "select selfoid from StockQuote");

   while ( !StockMarket.done ) {
    Enumeration e = vql.execute
     (0, 0, Constants.IRLOCK, Constants.RLOCK);

    System.out.println ();
    System.out.println ("Stock Market Report");
    while ( e.hasMoreElements() ) {
     StockQuote quote = (StockQuote) e.nextElement ();
     System.out.println (quote.symbol + ": $" +
               quote.price / 100.0);
    }
    System.out.println ();
    session.commit ();

    try { sleep (1000); } catch (InterruptedException ex) {}
   }
   session.endSession ();
  }
```

```
}

class StockMarketWriter extends Thread
{
 public void run ()
 {
  TransSession session = new TransSession (StockMarket.database);
  VQLQuery vql = new VQLQuery
    (session, "select selfoid from StockQuote where symbol = $1");

  for (int i = 0; i < 200; i++) {
   String symbol = StockMarket.randomSymbol ();
   vql.bind (symbol);

   Enumeration e = vql.execute
    (0, 0, Constants.IWLOCK, Constants.WLOCK);
   StockQuote quote = (StockQuote) e.nextElement ();

   int delta = StockMarket.random.nextInt() % 200;
   quote.price += delta;

   System.out.println ("Market update: " + symbol +
             (delta >= 0 ? " up " : " down ") + Math.abs
             (delta) / 100.0);
   session.commit ();
  }

  session.endSession ();
  StockMarket.done = true;
 }
}

class StockQuote
{
 String symbol;
 int price;
}
```

Multiple Threads, Single Session (MT1S)

In this model, multiple threads share a single session. This model also applies to an application that has multiple sessions with one of these sessions being shared by multiple threads.

MT1S is more difficult to use because the application must perform its own synchronization, in addition to the synchronization provided by the database.

How to use the MT1S model

This section presents two complete examples that use the MT1S model. In the MT1S model, multiple threads share a single session instance. The application needs to explicitly synchronize the execution of these threads to coordinate their activities.

When using an MT1S programming model in your application, multiple threads can read and write objects simultaneously. However, you should perform session related operations such as `commit()` or `rollback()` in a single thread, when no other threads accessing the objects in that session are active. This is the key restriction in a MT1S application – one thread should not read or write objects while another thread is, say, committing or performing a query.

The first example illustrates another stock market simulation. It consists of three threads: a main thread, a writer thread and a reader thread.

The main thread populates the database in a session with `StockQuote` objects that contain a ticker symbol and a price for a market security (stock). It then spawns a reader and writer thread, passing in the session used in the main thread.

The reader thread is responsible for printing out a stock market report for a set of stocks. The writer thread updates the price of a single stock at a time by randomly changing its price. Both the reader and writer threads operate in the same (shared) session opened in the main thread.

The reader and writer thread coordinate their activities so that the reader thread waits for a notification from the writer thread before attempting to print out a new stock report. The writer thread updates the price of a stock in a synchronized block, then commits the session and notifies the reader thread so that it can resume executing. Note that the reader thread prints out the stock report in a synchronized block, synchronizing on the same variable that synchronizes the writer thread. This synchronization guarantees that when the writer thread is committing changes to the database by invoking the `commit()` method, the reader thread is not accessing a stock quote.

The reader thread uses a VQL query to obtain all `StockQuote` instances, placing an `RLOCK` on the instances. The writer thread also uses a VQL query to obtain a certain `StockQuote` instance, placing a `WLOCK` on the retrieved instance. There is no lock contention between these two threads as they operate in the same session. The kernel handles locking of persistent objects on a per-session basis. Hence, if the reader thread first executed its query obtaining an `RLOCK` on the `Stock-`

Quote, and then the writer thread executed the query requesting a WLOCK, since both threads are in the same session, the lock would be upgraded when the writer thread executes.

Conversely, if the writer thread obtained a WLOCK on a StockQuote instance first, the reader thread would also be able to obtain the StockQuote instance. The request for an RLOCK on the instance from the reader thread is simply ignored because the session already possesses a higher strength WLOCK on that instance. By locking StockQuote instances during queries, it is ensured that:

- The reader sees a consistent snapshot of quotes during a transaction since it read-locks all instances that are read
- The writer cannot deadlock even if there were multiple writer threads updating the same instances, since the query write-locks all retrieved instances.

Notice that the first operation done in the run() loop of both the reader and writer threads is a setSession() method. When using a session in any thread apart from the one in which the session was created, the using thread must do an explicit setSession() on the session being shared. It is also desirable to do a leaveSession() when the thread finishes with the use of the session, so that the session can be successfully ended elsewhere.

For example:

```
import com.versant.fund.*;
import com.versant.trans.*;
import java.util.*;

public class StockMarket
{
 public static void main (String[] args) throws InterruptedException
 {
  if (args.length != 1) {
   System.out.println ("Usage: java StockMarket <database>");
   System.exit (1);
  }
  database = args [0];

  TransSession session = new TransSession (database);
  for (int i = 0; i < symbols.length; i++) {
   StockQuote quote = new StockQuote ();
   quote.symbol = symbols [i];
   quote.price = Math.abs(random.nextInt()) % 10000;
   session.makePersistent (quote);
  }
  session.commit ();

  new StockMarketReader (session).start ();
  new StockMarketWriter (session).start ();
 }

 static String randomSymbol ()
 {
  int index = Math.abs(random.nextInt()) % symbols.length;
  return symbols [index];
 }

 static String database;
 static boolean done = false;
 static Random random = new Random ();
 static String[] symbols = { "vsnt", "sunw", "ibm", "msft", "orcl",
               "hwp", "cpq", "dell", "sgi", "ifmx" };
}

class StockMarketReader extends Thread
{
 TransSession session;
 StockMarketReader (TransSession s)
 {
  session = s;
 }
 public void run ()
 {
  session.setSession ();
  VQLQuery vql = new VQLQuery (session, "select selfoid from
                 StockQuote");
  while (! StockMarket.done ) {
   Enumeration e = vql.execute (0, 0, Constants.IRLOCK,
                  Constants.RLOCK);
   synchronized (session) {
    System.out.println ();
    System.out.println ("Stock Market Report");
    while ( e.hasMoreElements() ) {
     StockQuote quote = (StockQuote) e.nextElement ();
```

```
      System.out.println (quote.symbol + ": $" +
                quote.price / 100.0);
     }
     System.out.println ();
     try { session.wait (); } catch (InterruptedException ex) {}
    }
   }
   session.leaveSession ();
  }
 }

 class StockMarketWriter extends Thread
 {
  TransSession session;
  StockMarketWriter (TransSession s)
  {
   session = s;
  }
  public void run ()
  {
   session.setSession ();
   VQLQuery vql = new VQLQuery
    (session, "select selfoid from StockQuote where symbol = $1");
   for (int i = 0; i < 20; i++) {
    String symbol = StockMarket.randomSymbol ();
    vql.bind (symbol);
    Enumeration e = vql.execute
     (0, 0, Constants.IWLOCK, Constants.WLOCK);
    StockQuote quote = (StockQuote) e.nextElement ();
    int delta = StockMarket.random.nextInt() % 200;
    synchronized (session) {
     quote.price += delta;
     System.out.println ("Market update: " + symbol +
               (delta) >= 0 ? " up " : " down ") +
               Math.abs (delta) / 100.0);
     session.commit ();
     session.notify ();
    }
    try { sleep (2000); } catch (InterruptedException ex) {}
   }
   StockMarket.done = true;
   session.leaveSession ();
   synchronized (session) {
    session.notify ();
   }
  }
 }

 class StockQuote
 {
  String symbol;
  int price;
 }
```

The second example illustrates a game score reporting application. It contains of a main thread that populates the database with `Game` objects. A `Game` contains a pair of team names and a pair of scores representing how many runs each team has scored. All the threads in the application share the session initially opened in the main thread.

For each game (which has a pairing of teams), the main thread spawns an updating thread, passing each thread the shared session. Each updating thread obtains a separate `Game` instance from the database using a VQL query, with a `WLOCK` on the `Game` instance selected. The updating thread then sets the in-progress scores of the on-going game between the two teams, and prints out the scores.

The main thread waits for all the updating threads to finish executing and then does a session `commit()` to write the in-progress scores of the games to the database. The main thread again spawns reporting threads to print out the final scores.

For each game, the main thread spawns a reporting thread, passing each thread the shared session. Each reporting thread obtains a `Game` instance from the database using a VQL query, with a `WLOCK` on the `Game` instance selected. The reporting thread then sets the final scores of the game between the two teams, and prints out the final scores.

The main thread waits for all the updating threads to finish executing, and does a session `commit()` to write the final scores to the database. This example is representative of an application model where there are multiple "worker" threads operating on a single session. The updating and reporting threads form the worker threads. The execution of these worker threads is coordinated from a main thread that initially spawns them together, waits for each to complete, and then performs database operations (such as a commit) on the shared session when no other threads are using the shared session. Notice that although the worker threads are simultaneously active writers (and readers), they operate on different instances and the database operation of a `commit()` is performed from a central coordinating thread.

For example:

```
import com.versant.fund.*;
import com.versant.trans.*;
import java.util.*;

public class NLGames
{
 public static void main (String[] args) throws InterruptedException
 {
  if (args.length != 1) {
   System.out.println ("Usage: java NLGames <database>");
   System.exit (1);
  }
  database = args [0];

  TransSession session = new TransSession (database);
  for (int i = 0; i < teams.length / 2; i++) {
   Game game = new Game ();
   game.team1 = teams [i*2];
   game.team2 = teams [i*2 + 1];
   session.makePersistent (game);
  }
  session.commit ();

  Thread[] update = new Thread[teams.length/2];
  Thread[] report = new Thread[teams.length/2];

  int i;
  for (i = 0; i < teams.length/2; i++) {
    update[i] = new ScoreUpdate (session, teams[i*2]);
    report[i] = new ScoreReport (session, teams[i*2]);
  }

  for (i = 0; i < update.length; i++) {
    update[i].start ();
  }

  for (i = 0; i < update.length; i++) {
    update[i].join ();
  }

  System.out.println ();
  session.commit ();

  for (i = 0; i < update.length; i++) {
    report[i].start ();
  }

  for (i = 0; i < update.length; i++) {
    report[i].join ();
  }

  session.endSession ();
 }

 static String database;
 static boolean done = false;
 static Random random = new Random ();

 static String[] teams = { "San Diego Padres",
              "Los Angeles Dodgers",
              "San Francisco Giants",
              "Colorado Rockies",
              "Atlanta Braves",
              "New York Mets",
              "Philadelphia Phillies",
              "Montreal Expos",
              "St. Louis Cardinals",
              "Chicago Cubs",
              "Houston Astros",
              "Cincinnati Reds" };
 }

class ScoreUpdate extends Thread
{
 TransSession session;
 String    teamName;

 ScoreUpdate (TransSession s, String aName)
 {
```

```
   session = s;
   teamName = aName;
  }

  public void run ()
  {
   session.setSession ();

   VQLQuery vql = new VQLQuery (session,
      "select selfoid from Game where team1 = $1");
   vql.bind (teamName);
   Enumeration e = vql.execute (0, 0, Constants.IWLOCK,
                  Constants.WLOCK);

   while ( e.hasMoreElements() ) {
     Game game = (Game) e.nextElement ();
     game.score1 = Math.abs (NLGames.random.nextInt()) % 4;
     game.score2 = Math.abs (NLGames.random.nextInt()) % 5;
     System.out.println ("Game Update: " + game.team1 + ": " +
               game.score1 + " " + game.team2 + ": " +
               game.score2);
   }
   session.leaveSession ();
  }
 }

 class ScoreReport extends Thread
 {
  TransSession session;
  String    teamName;

  ScoreReport (TransSession s, String aName)
  {
   session = s;
   teamName = aName;
  }

  public void run ()
  {
   session.setSession ();
   VQLQuery vql = new VQLQuery (session, "select selfoid from Game
                  where team1 = $1");
   vql.bind (teamName);
   Enumeration e = vql.execute (0, 0, Constants.IWLOCK,
                  Constants.WLOCK);

   while ( e.hasMoreElements() ) {
     Game game = (Game) e.nextElement ();

     game.score1 += Math.abs (NLGames.random.nextInt()) % 4;
     game.score2 += Math.abs (NLGames.random.nextInt()) % 5;
     if (game.score1 == game.score2) game.score2++;


     System.out.println ("Final Score: " + game.team1 + ": " +
      game.score1 + " " + game.team2 + ": " +
      game.score2);
   }
   session.leaveSession ();
  }
 }

 class Game
 {
  String team1;
  int score1;
  String team2;
  int score2;
 }
```

Life Cycle of a Persistent Object

The life cycle of a persistent Java object differs from that of a "regular," non-persistent Java object. A non-persistent object is constructed, exists for some period of time, and is then garbage collected (or the virtual machine terminates). The same is essentially true for a persistent Java object; however, there are some important differences:

- Since a persistent Java object is associated with an object in the database, its lifetime can extend across the execution of multiple virtual machines. In-memory garbage collection of a persistent Java object does not imply deletion from the database.

- A persistent Java object exists in one of several different states. When you access a persistent object or invoke the APIs of Transparent JVI, the state of the persistent object changes. These states and state transitions are described in detail below.

Persistent Object States

Broadly speaking, there are two basic states of a First Class Object in Transparent JVI: "transient" and "persistent." However, these high-level states can be described in more detail. For example, some persistent objects are "clean" (have not been modified) while others are "dirty" (have been modified). The various states of a transient or persistent object are summarized in the table below:

State of Object	Description
Clean	A persistent object that has not been modified in the current transaction.
Deleted	A persistent object that has been deleted in the current transaction. This deletion will become permanent if the transaction commits.
Dirty	A persistent object that has been modified in the current transaction.
Inactive	An object that has not been accessed in the current transaction. When the fields of an inactive object are accessed, it is "activated" by fetching the fields from the corresponding database object.
New	An object that has become persistent in the current transaction. If the current transaction rolls back, then this object will revert back to a transient state.
Transient	A non-persistent object. This could be a Persistent Capable object that has not yet become persistent, or an object that is not persistent capable at all.
Transient with identity	A transient object that possesses a LOID. This typically occurs when a persistent object is de serialized, or explicitly made transient.

Compatible Persistence Categories

Not all states are compatible with all persistence categories (see the section "Persistence Categorization"). As an example, Persistent Always objects can never be "transient". Similarly, Not Persistent Aware objects can never be in any of the persistent states, such as "clean" or "inactive." Entries in the following table that are marked as "X" indicate that the given combination is possible.

State of Object	Not Persistent Aware	Persistent Aware	Persistent Capable	Persistent Always
Clean			X	X
Deleted			X	X
Dirty			X	X
Inactive			X	X
New			X	X
Transient	X	X	X	
Transient with identity			X	

Properties of Persistent Object States

In addition to possessing a well-defined state, objects in JVI can have various properties as well. For example, persistent objects always have an associated session and database. However, these properties are not defined for all states of an object. The table below lists the various properties that are defined for each of the states:

States	Has LOID?	Has Session?	Is Active?
Clean	X	X	X
Deleted	X	X	
Dirty	X	X	X
Inactive	X	X	
New	X	X	X
Transient			
Transient with identity	X		

LOID

All persistent objects have a unique object identity, called the LOID. However, transient objects can also have a LOID.

See also “Object Identity” on page 180.

Session

A distinguishing property of persistent objects is that they have an associated session; transient objects do not. The session is the "context" in which the persistent objects exist. A persistent object must always be associated with exactly one session.

Active

A persistent object is active if its fields are cached in Java memory. If the fields of an inactive object are accessed (whose fields are not cached) they must be fetched from the database server. This process is called "activation". An active object can be "clean", "dirty" or "new". Deactivation occurs when the fields of a persistent object are flushed from the Java cache, and the object moves into one of the inactive states. For example, all persistent objects are deactivated when the associated transaction commits or aborts.

State Transitions

A persistent object changes state in response to various APIs or user actions. For example, immediately after a transaction commits (via the `TransSession.commit()` method), any cached persistent objects are changed to the "inactive" state. This means that the fields of the persistent objects are not cached - these objects are essentially "hollow". If you then read a field of a persistent object, this object will change from "inactive" to "clean". If you subsequently modify the object, it changes from "clean" to "dirty". (Of course, these actions also trigger other effects, such as objects being shipped to and from the database server and locks being placed on objects.)

The following table describes most of the possible state transitions in Transparent JVI. For instance, you can see that the commit operation takes objects in the "clean" state to the "inactive" state, and that accessing the fields of an "inactive" object puts it in the "clean" state.

From State	To State	Method / Action
clean	deleted	`deleteObject`
clean	dirty	`putfield (field modification)`
clean	inactive	`commit, rollback, releaseObject`
clean	non-existent	`garbage collection`
deleted	inactive	`commit, rollback`
dirty	clean	`checkpointCommit, commitAndRetain, rollbackAndRetain, groupWrite, query (flush)`
dirty	deleted	`deleteObject`
dirty	inactive	`commit, rollback, releaseObject`
inactive	clean	`getfield (field access), groupRead`
inactive	deleted	`deleteObject`
inactive	dirty	`putfield (field modification)`
inactive	non-existent	`garbage collection`
inactive	transient with identity	`makeTransient`
new	clean	`checkpointCommit, commitAndRetain`
new	transient	`rollback, rollbackAndRetain`
transient	new	`makePersistent`
transient	non-existent	`garbage collection`
transient	transient with identity	`setOid`
transient with identity	clean	`setPersistentState(Constants.CLEAN_STATE)`
transient with identity	dirty	`setPersistentState(Constants.DIRTY_STATE)`
transient with identity	inactive	`setPersistentState(Constants.INACTIVE_STATE)`
transient with identity	new	`setPersistentState(Constants.NEW_STATE)`

Explicit State Management

In the normal course of operations - accessing and modifying persistent objects, performing queries, beginning and ending transactions, and so on - the states and state transitions defined above yield "normal" expected behavior. That is, these persistent object states were designed to allow you to focus on the application domain rather than the details of object persistence. This is a cornerstone of "transparent persistence".

However, there are times when it is useful or necessary to explicitly manage the state of a Persistent Capable object. Transparent JVI provides the following methods of the `TransSession` class for performing explicit state management of persistent objects:

Methods	Description
`makePersistent`	Transforms a "transient" object into a "new" persistent object, by assigning it a new LOID and associating it with a session and a database.
`makeTransient`	Changes a persistent object to "transient with identity". The object is removed from the session's object cache and disassociated from its session and database. This transient object retains its previous LOID, making it easier to subsequently returned to a persistent state.
`setPersistentState`	Allows a "transient with identity" object to directly transition into a clean, dirty, inactive, or new state, associating it with a session and database. The `setPersistentState()` method is essentially the inverse of `makeTransient()`.

Object Identity

Objects that are stored in the database are either First Class Objects (FCOs) or Second Class Objects (SCOs). (See the sections "First Class Objects" and "Second Class Objects" in this chapter.) An FCO has its own object identity, or LOID, whereas an SCO is embedded within a containing FCO.

Object identity in FCOs

The identity of an FCO is linked to the identity of the corresponding database object.

Transparent JVI assures that there is never more than one FCO for a single database object. Therefore, `fco1 == fco2` if and only if `fco1` and `fco2` represent the same database object.

This case is only true for objects in the same session. Different sessions can and do have different FCOs for the same database object.

If the FCO does not define its own hash code, Transparent JVI assigns it a hash code that does not change in the lifetime of the database object.

Object identity in SCOs

SCOs have a separate identity in the Java Virtual Machine, but do not have a separate identity in the database. The database identity of an SCO is linked with the owning FCO.

Whereas an FCO is valid for the life of the associated session, an SCO is valid only in a single transaction. After a transaction ends, an SCO that existed during the transaction is no longer owned by an FCO. Accessing the field that referred to the SCO gives a new SCO with a different identity in Java.

Objects identity in transient objects

As mentioned above, a transient object can have an associated LOID. We describe such an object as being in the "transient with identity" state. Transient objects with identity allow persistent objects to be accessed without an associated session. These transient objects behave just like ordinary Java objects; however, they retain their persistent identities. Therefore, they can easily be reincorporated back into the Java cache at a later time by associating them with a session and database. This is done using the `TransSession.setPersistentState()` method. This is illustrated in the following example:

```
// Retrieve a persistent object from the database. A query
// or direct navigation could also be used. p can be
// referenced only in the context of the given session.
Person p = (Person) session.findRoot ("Principal");

// After calling makeTransient, p can be referenced like an
// ordinary Java object, without the session context.
session.makeTransient (p);

// ...

// Now p is reincorporated into the session cache, retaining
// its previous identity.
TransSession.setPersistentState (p, Constants.DIRTY_STATE);
```

Managing object identity

The persistent identity of an object is defined by its LOID. The Versant ODBMS assigns a new, unique LOID when a transient object becomes persistent -- by the `makePersistent()` or `makeRoot()` method, or through "transitive persistence". Many JVI applications do not need to explicitly manage object identity. However, Transparent JVI provides the following methods for explicitly managing the LOID of a persistent object:

Methods	Description
`getOidAsLong`	Returns the LOID of a persistent (or transient with identity) object as in 8-byte integer. This is the most efficient representation of a LOID in JVI.
`getOidAsString`	Returns the LOID of a persistent (or transient with identity) object as a string. This string is formatted as a "dotted decimal," such as "12.0.62".
`setOid(long)`	Sets the LOID of a transient object using the 8-byte representation. As a result, the object enters the "transient with identity" state.
`setOid(String)`	Sets the LOID of a transient object using the string representation.

Serialization

The Java serialization mechanism is used in Transparent JVI for:

- Serialization of SCO attributes
- Serialization of persistent objects

Serialization of SCO attributes

When an attribute of a persistent object does not map naturally to a Versant attribute type, it is serialized. If an error occurs during serialization, a `VException` of type `EVJ_SERIALIZATION_EXCEPTION` or `EVJ_DESERIALIZATION_EXCEPTION` is thrown. These are "wrapper" exceptions that contain the original exception (generally `java.io.IOException`) within. This is necessary because the original exceptions do not necessarily extend `java.lang.RuntimeException`.

Serialization of persistent objects

Persistent objects that implement `java.io.Serializable` can be serialized with the regular Java serialization mechanism. In general, serialization of a persistent object gives the same result as serialization of regular, non-persistent objects. However, when serializing a persistent object, the executing thread must be associated with the correct session. Serializing an inactive object causes the object to be fetched from the database and locked.

Serialized persistent objects retain their database identity. Every persistent object has an extra LOID field, which is automatically serialized as part of the object. Therefore, when a persistent object is de serialized, it has the same LOID as the original, serialized object. However, the object does not immediately become persistent upon de serialization. Instead, it enters the "transient with identity" state. This allows you to choose the appropriate state for the persistent object – dirty, clean, new, or inactive – based on the application requirements. Use the `TransSession.setPersistentState()` method to make the object persistent.

Note that the serialized format of a persistent object is compatible between enhanced and unenhanced classes. That is, you can enhance a class (as Persistent Capable), serialize an instance of this class, and then deserialize it using the original, unenhanced class. The same is true for the inverse, starting with the unenhanced class. Hence remote clients (using, say, CORBA, EJB, or RMI) can interact with persistent objects without requiring a JVI environment.

Cloning

Persistent objects that implement the `java.lang.Cloneable` interface can use the Java cloning mechanism. When a persistent object is cloned, the resulting object is a shallow copy of the original object. This shallow copy does not retain the database identity of the original object, and is "transient".

Transient Fields

When a field of a persistent-capable class is marked with the Java keyword “transient”, this field will not be stored in the database. Since this field is transient, it is part of the transient, runtime state of the object, but not its persistent state. In JVI Transparent Binding, transient fields are separated into two categories, pure transient attributes and computed transient attributes.

Pure Transient Attributes

Pure transient attributes are completely independent from any persistent state or identity in a persistent Java object. In this respect, they are similar to static (class) attributes. Accessing a pure transient attribute does not cause the corresponding Versant object to be fetched from the database and does not place any locks on the object. The Versant 'flush' mechanism does not 'null' pure transient attributes. But the attributes could point to a non-existent object due to, for e.g., garbage collection occurring in the cache.

When you call `commit`/ `commitAndRetain`/ `commitAndCleanCod`, persistent objects are flushed from Java-cache and C-cache and Java proxy remains in Java-cache. This proxy object keeps transient value. When you call `System.gc()`, the proxy is removed from Java cache and on accessing persistent object once again after `System.gc()`, JVI has to re-read object to Java-cache. At this point, JVI has no way of knowing the value of pure transient attribute. The pure transient attribute is set to false (default). This is where the usage of transient objects and persistent objects makes a difference because persistent object is under transaction control when a transient object is not under transaction control.

Pure transient attributes are useful for storing certain types of “meta” information associated with persistent objects. It is completely the responsibility of your application to keep a pure transient attribute synchronized with the identity and/or state of a persistent object. One way to keep a pure transient attribute properly maintained is to use the persistent object hooks, which give you notification of key moments in a persistent object’s life cycle.

See also “Persistent Object Hooks” on page 185.

Computed Transient Attributes

Computed transient attributes behave very much like persistent attributes, except that they are not stored in the database. Instead, it is anticipated that a computed transient attribute will be used in conjunction with the persistent object hooks, such as `postRead()` and `preWrite()` (see “Persistent Object Hooks” in this chapter). These hooks allow the computed transient attributes to be set based on the contents of one or more persistent fields.

As mentioned, computed transient attributes are similar to persistent attributes in many ways:

- Reading a computed transient attribute causes the corresponding Versant object to be fetched from the database (if necessary), and places a default lock (generally a read lock) on this object. The `preRead()` and `postRead()` hooks will be invoked while reading the database object.

- Modifying a computed transient attribute marks the object as dirty, and places a write lock on the corresponding database object. Dirty objects can be written to the Versant server (for example, on transaction commit).
- Computed transient attributes are set to null (for reference attributes) when the persistent Java object is invalidated - that is, when the object is flushed from the Java cache. This allows the data, which is no longer valid, to be garbage collected.

Configuration

By default, a Java field marked with the “transient” keyword is a computed transient attribute. However, the `-puretransient` option to the Enhancer (`com.versant.Enhance`) and Launcher (`com.versant.Launch`) can be used to override the default for fields marked as “transient”.

If the Enhancer is invoked with the `-puretransient` option, then for all persistent classes that are enhanced, fields marked with the Java “transient” keyword will be pure transient attributes rather than the default computed transient. The Launcher also accepts the `-puretransient` option, whereas the `-puretransient` option can be used selectively on a class-by-class basis with the Enhancer (by invoking the enhancer more than once), this option pertains to all persistent classes that are enhanced by the Launcher.

Persistent Object Hooks

The JVI Transparent Binding provides hooks that can notify an application at key moments in the life cycle of a persistent object. The application uses these hooks by implementing certain methods on the persistent object itself. The system will automatically invoke these methods when the persistent object transitions from one state to another. There are eight methods on a persistent object that can be overridden:

Methods	Description
`activate`	Called when the state of the object enters the Java cache. The `activate` method is typically used to register the persistent object with its environment.
`deactivate`	Called when the state of the object leaves the Java cache. The `deactivate` method is typically used to unregister the persistent object from its environment. The hook will be invoked only when the handles/Object references in Object cache are valid.

`preRead`
Called before the state of the object is read from the database. The `preRead` method is typically used to free resources associated with transient fields before the read operation.

`postRead`
Called after the state of the object is read from the database. The `postRead` method is typically used to initialize the transient fields in the object based on the persistent fields.

`preWrite`
Called before the state of the object is written to the database. The `preWrite` method is typically used to initialize the persistent fields in the object based on the transient fields.

`postWrite`
Called after the state of the object is written to the database. The `postWrite` method is typically used to free resources associated with persistent fields after the write operation.

`vDelete`
Called when a persistent object is deleted (via the `TransSession.deleteObject()` method). The `vDelete` hook is typically used to delete any associated “subordinate” objects.

`vMaterialize`
Called when the Java representation of a persistent object is first instantiated, or “materialized.” The `vMaterialize` hook is typically used to allocate resources for the in-memory representation of a persistent object.

The system provides empty "do-nothing" implementations of each method that will be used unless the method is explicitly overridden in the class of the persistent object. The default, empty method will only be overridden if the type signature is correct. If there is an mistake in the type signature, the method will not be invoked, and no error will be generated.

The type signatures must be exactly as defined below:

```
public void activate   ()
public void deactivate  ()
public void preRead   (boolean activate)
public void postRead   (boolean activate)
public void preWrite   (boolean deactivate)
public void postWrite  (boolean deactivate)
public void vDelete   ()
public void vMaterialize ()
```

The `preRead` and `postRead` methods have a parameter that indicate whether the object will be activated after the read operation. The `preWrite` and `postWrite` methods have a parameter that indicate whether the object has been deactivated. The `activate` and `deactivate` methods have no parameters.

The following shows which methods are invoked for some typical operations on persistent objects:

Create a new persistent object (`makePersistent` or transitive persistence)	activate
Accessing (reading or writing) a previously inactive object	preRead (true) read the object from the database postRead (true) activate
Refresh an object (`refreshObjects`)	preRead (false) read the object from the database postRead (false)
Write a modified object to the database (`VQLQuery`, `select`, `checkpointCommit`, `commitAndRetain`, or `groupWriteObjects`)	preWrite (false) write the object to the database postWrite (false)
Commit a modified object	deactivate preWrite (true) write the object to the database postWrite (true)
Commit an unmodified, but active, object	deactivate
Delete an active object	deactivate
Rollback an active object	deactivate

Example Uses of the Persistent Object Hooks

Perhaps the best way to understand the hooks defined above is to consider example that uses for them.

Compression and Decompression

A simple example of using the postRead and preWrite hooks is transparent compression and decompression of data, such as text or images, in a persistent object. Compressed data can be stored in the database, and automatically uncompressed when the object is read from the database. If the object is modified, the data can be automatically compressed when written to the database.

In this scenario, there are two fields representing the data that will be compressed. A persistent field contains the compressed data, since this information is written to the database. A transient field holds the uncompressed data. The postRead method is implemented to decompress the data in persistent field, storing the result in the transient field. The application accesses the uncompressed data in the transient field directly. The preWrite method is implemented to compress the data in the transient field, storing the result in the persistent field. Only the compressed data held in the persistent field is written to the database.

In the most straightforward implementation, the application is not aware of any compression or decompression. Depending on the memory and performance requirements, more advanced implementation could be developed by more directly involving the application. For example, by implementing a getData method, the data could be uncompressed "on demand", as needed by the application. The preWrite method would still be useful for compressing the data before writing the object to the database.

Event Notification Registration

The activate and deactivate methods are typically used to register the object within its environment. An example of this is registering with the Event Notification system to be notified when another transaction has modified objects that are active within the current session. The application can then take the appropriate steps to update the view of these objects (for example, by refreshing the objects and redisplaying the results). This is a realistic scenario in many applications that do not maintain locks for persistent objects, possibly using the optimistic locking mechanism of the Versant ODBMS. By registering objects in the activate method and unregistering in the deactivate method, the application can filter out notifications for objects that are not active in the current session.

Special Notes

Garbage Collection of Active Objects

If an object has been activated but has not been modified in the current transaction, it can be garbage collected by the system. If this happens, the object will not receive a deactivate notification. If the application always requires a `deactivate` notification for any active object, then the application must ensure that the object is not garbage collected by maintaining a reference to the object.

Exceptions While Committing a Transaction

When an object is committed to the database, the deactivate method is invoked on the object before the commit actually occurs. If, for whatever reason, the commit operation fails and generates an exception, the object is not actually deactivated (that is, the state of the object is not flushed from the cache). Since the deactivate method has already been called when the commit fails, the system invokes the `activate` method. From the perspective of the application, the object is deactivated and activated in one operation; in reality, however, the state of the object remains in the cache the whole time.

Example Program

The following example implements the persistent object hooks by simply printing out a message. It performs several operations on a persistent object, causing the hooks to be invoked.

```
import com.versant.fund.*;
import com.versant.trans.*;

public
class Hooks
{
  public static
  void main (String[] args)
  {
    if (args.length != 1) {
      System.out.println ("Usage: java Hooks <database>");
      System.exit (-1);
    }

    String    database = args [0];
    TransSession session = new TransSession (database);
    Person    person  = new Person ();
    Object[]   objects = { person };

    // A newly created persistent object will be activated.
    // prints "activate"
    session.makePersistent (person);
```

```
// Commit causes the object to be deactivated and written.
// prints "deactivate"
// prints "preWrite(true)"
// prints "postWrite(true)"
session.commit ();

// When the object is first touched, its state is read
// into the cache, and the object is activated.
// prints "preRead(true)"
// prints "postRead(true)"
// prints "activate"
person.name = "John Doe";

// The state of the object is still in the cache, so
// again no hooks are invoked.
person.age = 25;

// Checkpoint commit causes the object to be written, but
// not released from the cache. Hence postWrite is called.
// prints "preWrite(false)"
// prints "postWrite(false)"
session.checkpointCommit ();

// Since nothing has changed, no hooks are invoked.
session.checkpointCommit ();

// The state of the object is still in the cache, so
// again no hooks are invoked.
person.age = 26;

// By default, selecting causes all instances of the class
// to be flushed. The object is not released from the cache,
// so postWrite is called.
// prints "preWrite(false)"
// prints "postWrite(false)"
session.select (Person.class, null);

// The state of the object is still in the cache, so
// again no hooks are invoked.
```

```
person.age = 27;

// The object is flushed but not released, so it is similar
// to checkpointCommit or select flush.
// prints "preWrite(false)"
// prints "postWrite(false)"
session.groupWriteObjects (objects, 0);

// The state of the object is still in the cache, so
// again no hooks are invoked.
person.age = 28;

// The object was refreshed, but not activated (because it
// was already in the cache). Hence preRead is called.
// prints "preRead(false)"
// prints "postRead(false)"
session.refreshObjects (objects, database, Constants.RLOCK);

// The state of the object is still in the cache, so
// again no hooks are invoked.
person.age = 29;

// The object us directly released from the cache, so it
// is deactivated without writing any changes.
// prints "deactivate"
session.releaseObjects (objects);

// The object was already released, so no hooks are invoked.
session.releaseObjects (objects);

// The object had been released, so touching it causes it
// to be read into the cache and activated.
// prints "preRead(true)"
// prints "postRead(true)"
// prints "activate"
person.age = 30;

// The rollback causes the object to be released from the
// cache and deactivated.
// prints "deactivate"
session.rollback ();
```

```
    // The object was already released, so no hooks are invoked.
    session.rollback ();

    // The object had been released, so touching it causes it
    // to be read into the cache and activated.
    // prints "preRead(true)"
    // prints "postRead(true)"
    // prints "activate"
    person.age = 31;

    // The commit causes the object to be flushed and released
    // from the cache.
    // prints "deactivate"
    // prints "preWrite(true)"
    // prints "postWrite(true)"
    session.commit ();

    // The object had been released, so touching it causes it
    // to be read into the cache and activated.
    // prints "preRead(true)"
    // prints "postRead(true)"
    // prints "activate"
    person.age = 32;

    // The delete operation causes the object to be released
    // from the cache.
    // prints "deactivate"
    session.deleteObject (person);

    // The object had already been released, so the rollback
    // does not invoke any hooks.
    session.rollback ();

    session.endSession ();
  }
}

class Person
{
  String name;
  int  age;

  public void activate  () { System.out.println ("activate" ); }
  public void deactivate () { System.out.println ("deactivate"); }
  public void preRead  (boolean activate)
    { System.out.println ("preRead("  + activate  + ")"); }
  public void postRead  (boolean activate)
    { System.out.println ("postRead(" + activate  + ")"); }
  public void preWrite  (boolean deactivate)
    { System.out.println ("preWrite(" + deactivate + ")"); }
  public void postWrite (boolean deactivate)
    { System.out.println ("postWrite(" + deactivate + ")"); }
}
```

Class Loaders

A Java class loader is a special object that loads Java classes by name. The system class loader is the default class loader in the virtual machine. For example, when your application refers to the `java.lang.String` class, the system class loader searches the `CLASSPATH` for the class file or jar file containing the Java byte codes for the String class. (However, some Java virtual machines may use a different mechanism than the `CLASSPATH`.) The class loader reads in the byte codes and converts them into a Java class object.

Java also allows specialized class loaders to be created and used. For example, Java applets are typically loaded by a network class loader that loads Java classes from a URL (sometimes called the `CODEBASE`). In a Web server, Java servlets may also be loaded by a special class loader (this provides the ability to "reload" servlets without restarting the server).

Dealing with class loaders can be tricky, because a class loader defines its own class namespace. In other words, it is possible to load the same class twice in two different class loaders. It is even possible to load two different classes that have the same name (including package) using different class loaders. Having some knowledge of class loaders is useful when debugging problems with applets, servlets and other class loader environments.

Transparent JVI uses class loaders indirectly, when converting objects in the database to persistent Java objects. Using the name of the class, persistent objects are instantiated as necessary. Therefore, when using Transparent JVI in environments such as applets or servlets that load classes using a class loader (other than the system class loader), the following rules must be followed:

- The JVI system classes, such as `com.versant.TransSession`, should be loaded by the system class loader. This is normally the case when the JVI jar file is included in your `CLASSPATH`.

- Transparent JVI should be informed of the class loader (such as the network class loader) using the `TransSession.setSessionClassLoader()` method.

The Session Class Loader

Transparent JVI supports the concept of a session class loader. Each JVI session has an associated class loader that is used to load persistent-capable classes (and related classes such as the "pickler"). The following methods of the `TransSession` class allow you to get and set the session class loader:

```
ClassLoader getSessionClassLoader ();
void setSessionClassLoader (ClassLoader loader);
```

By default, a session's class loader is the system class loader. However, when using Transparent JVI with applets, servlets or other specialized class loading environments, you can use the `setSessionClassLoader()` method to change the default. You should call `setSessionClassLoader()` immediately after constructing the session (or after resetting the Java cache by the `clearJavaCache()` method).

You can generally obtain the appropriate class loader using the `Class.getClassLoader()` method. For example, assume that you are writing a servlet for a Web server that loads servlets through a `ServletClassLoader`. Then, within the servlet implementation, you can use the `Class.getClassLoader()` method to find the session class loader. This class loader should be passed to the `setSessionClassLoader()` method. The following example code demonstrates this:

```
import com.versant.trans.*;
import javax.servlet.*;
import javax.servlet.http.*;
class MyServlet extends HttpServlet
{
  void doGet ( HttpServletRequest request,
           HttpServletResponse response )
   throws ServletException, java.io.IOException
  {
   // Find out which class loader loads the servlet.
   ClassLoader loader = getClass().getClassLoader();
// Begin a session that uses the servlet's class loader.
   TransSession session = new TransSession ("mydatabase");
   session.setSessionClassLoader (loader);
// Here the servlet would use the session to
```

```
    // access the database.
    // ...
  }
}
```

Note that the example above works even if the servlet is loaded by the system class loader. In this case, the `getClassLoader()` method returns the null value. Passing null to the `setSessionClassLoader()` method sets the session class loader to the default, the system class loader.

Every session has its own session class loader. This means that different sessions can use (potentially) different class loaders. This is only possible because a session has its own private cache of persistent objects, which is not shared with other sessions. This feature can be useful in a server environment (such as a Web server or application server), so that persistent classes may evolve without restarting the server.

Object Sharing

JVI enables transparent sharing of C++ objects with Java programs. The sharing is enabled through generating "Java Foreign View" source files that correspond to the schema defined from C++. These `.java` source files must be compiled and enhanced with other classes in the Transparent JVI program.

The Foreign View approach has the following features:

- The object sharing is one-directional.
- The Transparent JVI program can access database schema for a class created from a language other than Java. Sharing the schema of a class defined from a Transparent JVI program with other languages transparently, is not addressed.
- Transparent sharing of the object attributes is supported, but not the methods.
- However, you can write methods in the generated source files of classes and then use these methods as you would in any other persistent Java class.

For more information, on object sharing and a step-by-step tutorial, please refer the on-line tutorial provided with your JVI installation.

Restrictions

Overview

You should be aware of the restrictions that exist in Transparent JVI in the following areas:

- Reflection
- Native methods
- `null` attributes
- Categorization of interfaces and superclasses

Reflection

Using the Java reflection mechanism on FCOs can create problems due to changes made by the enhancer. However, the JVI Enhancer “fixes” calls to the reflection APIs so that reflection on FCOs is supported. For this to work, the class that invokes any of the reflection APIs must be enhanced as “Persistent Aware” (or Persistent Capable or Persistent Always). The list of reflection APIs that are processed by the enhancer is:

- Class.getConstructors
- Class.getDeclaredConstructors
- Class.getDeclaredFields
- Class.getDeclaredMethods
- Class.getField
- Class.getFields
- Class.getMethods
- Class.getSuperclass
- Field.get*
- Field.set*
- Member.getModifiers

Native methods

The problems inherent with using reflection also apply to native methods. Therefore, native methods can invoke methods of an object, but cannot access its fields.

Null attributes

Certain Java fields map naturally to Versant attribute types, with the exception of the `null` value. The database cannot distinguish between the `null` value and an empty or zero value.

The following types cannot be distinguished:

- Arrays - an array of length zero is seen as a `null` value in subsequent transactions.
- Primitive wrappers (`java.lang.Integer` and so on.) - a null value is seen as zero (or false, in the case of `java.lang.Boolean`)

Categorization of interfaces and superclasses

If a class is categorized as Persistent Always, Persistent Capable or Transparent Dirty Owner, then its superclass must be given the same category.

If an FCO has an attribute that is an interface class, and the interface is categorized as Persistent Always or Persistent Capable, then the attribute can only hold objects whose class is Persistent Always or Persistent Capable.

If the interface is categorized as Persistent Aware or Not Persistent Aware, then the attribute can hold objects of any category.

Note that the enhancer does not modify interface classes.

Performance Tuning

To improve the performance of your application in Transparent JVI, you must be aware of:

- SCO tracking
- Arrays
- Serialization and Deserialization
- Versant collections
- Memory management
- Group operations

NOTE:- The performance of `groupRead()` API is highly improved from this version.

SCO tracking

SCO tracking provides complete transparency during SCO modification. When an SCO is modified, the owning FCO is automatically dirtied. This complete transparency, however, incurs additional runtime overhead.

An SCO table keeps track of the owner of each SCO in the following manner:

- Each time a reference to an SCO is given to the application code, the table is updated with the owning FCO.
- Each time an SCO is modified, a table lookup finds the owning FCO and marks it as dirty.
- When a transaction ends, the table is cleaned of all entries associated with that transaction.

One of the drawbacks of SCO tracking is that it decreases performance even for applications that do not use SCOs or need SCO tracking.

Providing the `-manualdirty` option to the enhancer can disable SCO tracking. The `-manualdirty` option disables automatic dirtying only for SCOs. If an FCO is changed directly, the FCO remains marked as dirty.

When SCO tracking is disabled, it may be necessary to manually dirty an owning FCO when an SCO is changed.

If the FCO is not manually dirtied, the change to the SCO is not stored in the database, unless the FCO is implicitly dirtied by directly changing one of its fields.

To manually dirty the FCO, use the `TransSession.dirtyObject()` method.

NOTE:- Classes categorized as Transparent Dirty Owner do not use the SCO table. Instead, they maintain a reference to the owner within the SCO itself. Instances of these classes are not affected by the `-manualdirty` option.

Arrays

When an FCO has a field that is an array of FCOs, performance can suffer due to the way Java handles arrays.

Arrays are SCOs; therefore, SCO tracking normally occurs when using arrays. The first time the array is accessed within a transaction, the entire array is converted from Versant links to FCOs. This action is unnecessary when only a small number of elements of the array is accessed in the transaction.

Serialization and Deserialization

When an attribute of an FCO does not map naturally to a Versant type, the object is serialized with the Java serialization mechanism. This serialization can result in a performance loss with large objects that contain references to many persistent objects. The first time this attribute is accessed within a transaction, the entire object is de-serialized. Each reference to an FCO is recreated and converts a Versant link to the FCO.

Java collections `java.util.Vector` and `java.util.Hashtable` are serialized. Performance typically improves when you use the corresponding `VVector` and `VHashtable` classes.

Versant Collections

Most of the Versant collections in Transparent JVI are implemented in the front-end. In the database, these collections are stored as simple “`vstrs`.” This action can have grave performance implications with large numbers of elements in a collection.

For example, consider the `VVector` class. Although `VVector.elementAt()` is a constant-time operation, accessing the vector with any method for the first time in a transaction is a linear-time operation. It is linear-time because the entire `vstr` of Versant links must be brought from the server to the client. This case also applies when you look up an element in a `VHashtable`, or insert an element in a `ListOfObject`.

NOTE:- One approach to avoid this problem is to use collections sparingly. For example, objects can be stored directly in the database and queried with VQL instead of being stored in a `VHashtable.`

The `LargeVector` class overcomes this problem. When stored in the database, `LargeVector` is broken into multiple "nodes". Only the needed nodes are brought to the client. This action increases the overhead for small collections, but drastically reduces the overhead for large vectors and can provide significant performance benefits.

Memory management

Memory management in Transparent JVI is mostly automatic. For optimal performance, however, it is important to be aware of memory management issues.

For each session, three basic areas of memory have to be managed:

- Java front-end cache
- C front-end cache
- COD Table

Java front-end cache

In Transparent JVI, persistent objects are cached in Java memory. This has two important consequences:

- If a cached object is subsequently accessed (by traversing references or by a query), the object is not fetched from the database. Instead, the cached copy of the object is used.
- No more than one copy per session of a persistent object exists in Java memory at one time. Therefore, the identity of a persistent object in Java is associated with the identity in the database.

Releasing objects from the Java cache

Persistent objects in the Java cache are either active or inactive.

- An active object has already been fetched from the database. It is not fetched again until it becomes inactive.
- An inactive object has not been fetched from the database. It is fetched upon the first access to one of its fields.

When an object is released from the Java cache, it changes state from active to inactive. When an object is fetched, it changes state from inactive to active.

Objects are typically released from the Java cache when a transaction ends, by either committing or rolling back the transaction. However, objects can be explicitly released with the `TransSession.releaseObjects()` method.

An object could be explicitly released for two reasons:

- Free up memory: when an object is released, its fields are set to null. This action can enable garbage collection by removing references to the objects referenced by the released object.
- Force an object to be fetched from the database the next time it is accessed: this action could allow changes committed by another transaction to be seen if no lock is kept on the object.

An object can be re-fetched immediately (as opposed to being re-fetched on the next access) with the `TransSession.refreshObjects()` method.

Garbage collection

Objects are removed, or cleared, from the Java cache with the standard Java garbage collector. When a persistent object is no longer referenced by the application program, it can be removed from the cache.

Only persistent objects that are "clean" can be garbage-collected. "Dirty" objects (objects that have been modified in the current transaction) are not removed from the cache until the changes are committed or rolled back.

Removing an object from the cache is different from releasing the object. Releasing an object merely changes its state to "inactive" and sets its fields to null.

Weak references

Transparent JVI maintains tables of persistent objects that are in the Java cache. These internal tables use a Java feature called weak references. Weak references are special objects that can reference other objects while still allowing the garbage collector to reclaim those objects being referenced.

Weak references can decrease performance, particularly in the JDK 1.1 release of the Java Virtual Machine. To work around this problem, Transparent JVI provides a mechanism to disable weak references and to manually remove objects from the cache.

To disable weak references in the Java cache, set the `javaCacheUsesWeakRefs` property to `false` when you construct a `TransSession` object.

To manually remove objects from the Java cache, use the `TransSession.clearJavaCache()` method, or one of its variants.

Manually removing an object from the cache is not correct if the application maintains and uses references to the object. If manual removal occurs, multiple Java objects could be instantiated for a single database object in a session, causing unpredictable behavior.

The SCO table in "SCO Tracking" also uses weak references. Weak references in the SCO table can be disabled with the `TransSession.autoDirtyUsesWeakRefs()` method.

C front-end cache

When a persistent object is fetched from the database, it is brought into the C front-end cache. From there, it is transferred into the Java cache. After an object is transferred into the Java cache, it is released from the C cache. Hence management of the C cache is generally not an issue.

COD table

Unfortunately the COD table (see description in Fundamental JVI) is not automatically managed. Therefore, applications with long-running sessions that access a large number of objects could need management of the COD table.

The simplest way to remove objects from the COD table is to use the `ransSession.commitAndCleanCod()` method. This method clears the COD table after the commit operation.

To manually remove objects from the COD table with `commit()`, use the `TransSession.zapObjects()` method.

Group operations

Deleting, reading, and writing multiple objects can be performed in a single request with the methods `TransSession.groupDeleteObjects()`, `TransSession.groupReadObjects()` and `TransSession.groupWriteObjects()`.

Group operations are generally more efficient than performing the same operations one at a time on multiple objects, since only one message is sent to the database back end.

Reflection Class Enhancement

The class com.versant.trans.Reflection has been enhanced to improve the JVI performance.

Field to Method mapping is introduced in this class and the default values for the getter map size; setter map size and the maximum map size are as follows -

```
getterMapSize = 100;
setterMapSize = 100;
MAX_MAP_SIZE = 500;
```

These values can also be overridden from the command line by passing these system parameters -

```
-DGETTERMAP_SIZE
-DSETTERMAP_SIZE
-DMAX_MAP_SIZE
```

E.g- `java -DGETTERMAP_SIZE=500 -DSETTERMAP_SIZE=500 -DMAX_MAP_SIZE=1000`

Debugging Internals

Overview

Using debuggers with programs that use Transparent JVI requires some knowledge of the way the `com.versant.Enhance` utility enhances the Java classes to enable persistence. This information should be useful when you step through the Java program using a debugger.

To explain enhancements and how it effects your application when you debug, this section describes:

- Enhanced Class Files for Persistent Classes
- Debugging Transparent JVI applications

Enhanced class files for persistent classes

For every class that is designated as Persistent Capable or Persistent Always, the enhancer modifies the class file, and generates one new class file. If the original class is called `Person`, then the enhancer modifies `Person.class` and adds `Person_Pickler_Vj.class`. The new `Pickler` has generated code for defining the database schema for the class. The class name is always of the form: `class-name_Pickler_Vj.class`.

Application algorithms do not change, but attributes of persistent classes (Persistent Capable or Persistent Always) are renamed and new attributes are added to persistent classes. These modifications alter the internal structure of these classes so that the state of the class can adapt with the persistence requirements.

To explain enhanced class files for persistent classes, this section describes:

- Inheritance of persistent classes
- Additional attributes
- Renamed attribute
- Hash code

Inheritance of persistent classes

Persistent classes are re-rooted to derive from one of the following classes:

```
com.versant.trans.Jod
com.versant.trans.Persistent
com.versant.trans.CapableWithHash
```

The latter two of these classes are both subclasses of `com.versant.trans.Jod`. `com.versant.trans.Jod` which contains flags to maintain the state of the persistent object.

These flags denote if the object:

- Can be read.
- Can be written to the database.
- Is dirty (its value has been modified).
- Is designated to become persistent on a commit.
- Is persistent, but not dirty.
- Has been deleted.

The `com.versant.trans.Jod` class also contains the `com.versant.fund.Handle` attribute of the persistent object.

The `com.versant.fund.Handle` class encapsulates the object LOID, which is a unique logical object identifier.

Additional attributes

As a result of the re-rooting, enhanced persistent classes inherit the following attributes from `com.versant.trans.Jod`, all prefixed by `"_vj_"`:

```
_vj_flags
_vj_handle
_vj_isUptodate
_vj_isDirty
_vj_classHandle
_vj_hashCode
```

These are internal variables to keep track of the state of persistent objects.

Privatized attributes

All attributes of a persistent object, unless static or transient, are changed to be `private`. This is to prevent other applications classes that are not designated as Persistent Aware from accidentally accessing persistent object attributes directly, bypassing JVI's persistence mechanisms.

Hash Codes

The hash code value of an object plays a significant role in identifying an object. The hashCode() method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

The general contract of the hashCode() method is that whenever it is invoked on the same object during an execution of a Java application, it must consistently return the same integer. This integer need not remain consistent from one execution of an application to another execution of the same application. If two objects are equal according to the equals() method, then calling the hashCode() method on each of the two objects must yield the same integer result.

When adding persistence to objects, the hash code value is also maintained persistently. Whenever the hashCode() method is invoked on the same persistent object across executions of a Java application, it consistently returns the same integer. To implement this property, JVI implements the three different superclass categories previously listed.

Persistent Always classes are automatically designated to become persistent at commit time, as soon as an instance of the class is created. Internally, this translates to creating a LOID for the instance when it is constructed. Since the LOID is a unique identifier, it can be reused as the hash code value for that object.

If, however, the user class designated as Persistent Always has already implemented the (overloaded) hashCode() method, then that function is used to determine the persistent hash code value. All Persistent Always classes thus derive from com.versant.trans.Persistent.

Objects of a Persistent Capable class could become persistent in the database at commit time, if they are named objects in the database, reachable through transitive persistence, or explicitly used in a makePersistent() method call. To deal with the hashCode() requirements, such classes are made to derive from com.versant.trans.CapableWithHash if they do not implement an overloaded hashCode function, in which case the hash code value is stored persistently. Classes are made to derive from com.versant.trans.Jod if they do implement an overloaded hashCode function.

Suppose that there is a simplistic Person class shown below as an example.

For example:

```
public class Person
{
  String name;
  int age;

  Person (String aName, int anAge)
  {
   name = aName;
   age = anAge;
  }

  public int age ()
  {
   return age;
  }

  public void increaseAge ()
  {
   age++;
  }
}
```

After enhancement, if you run the `javap` utility on the enhanced class file as follows,

```
javap Person
```

You can see that its superclass has been changed to `com.versant.trans.CapableWithHash`, and its two attributes, `name` and `age`, have been changed to `private`. New methods have been added as well.

Debugging Transparent JVI Applications

The following three fields inherited from `com.versant.trans.Jod` mainly reflects the state of a persistent object: `_vj_isUptodate`, `_vj_isDirty` and `_vj_Handle`.

Up-to-date flag

This boolean flag represents whether a persistent object needs to be fetched from the database.

Dirty flag

This boolean flag represents whether a persistent object has been modified and hence needs to be written to the database.

Handle

In Fundamental JVI, a `Handle` has a one-to-one relationship with COD in the Object Cache.

How to debug a Transparent JVI application

Using a modified version of the `CreatePerson` example in the Transparent JVI Tutorial, the following example shows the different states a Persistent Capable object, that goes through in a typical application:

- Transient
- Newly-persistent state (made persistent, but not committed)
- Committed
- Clean
- Modified

An example class, `CreatePersonModified`, is listed below. Note that the Class Person remains the same as in the Tutorial.

After starting a session, the following program creates a `Person` instance, makes the instance persistent, and commits the transaction. Then the `Person` object is then accessed, modified, and committed.

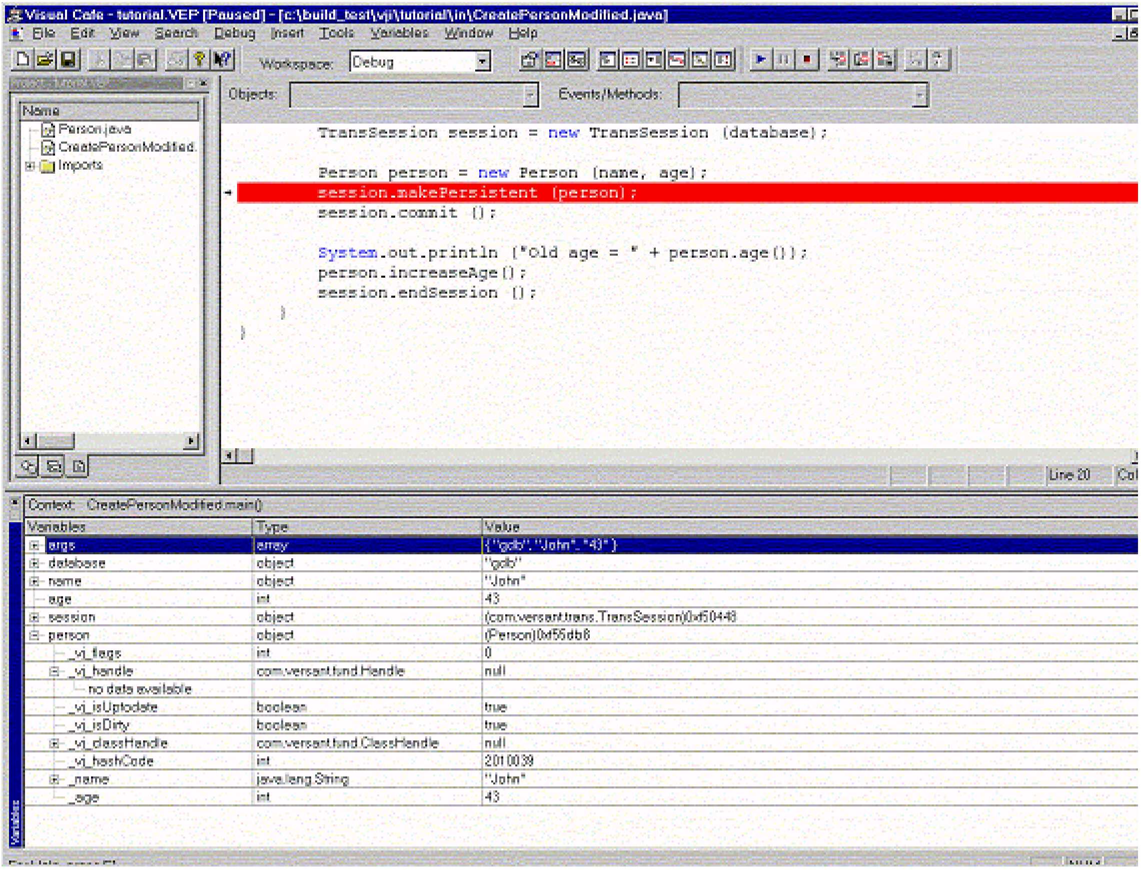
```
import com.versant.trans.*;

public class CreatePersonModified
{
 public static void main (String[] args)
 {
  if (args.length != 3) {
   System.out.println
    ("Usage: java CreatePerson <database> <name> <age>");
   System.exit (1);
  }
  String database = args [0];
  String name  = args [1];
  int age  = Integer.parseInt (args[2]);

  TransSession session = new TransSession (database);

  Person person = new Person (name, age);
  session.makePersistent (person);
  session.commit ();

  System.out.println ("Old age = " + person.age());
23  person.increaseAge();
24  session.endSession ();
25 }
26 }
```

Transient State

Look at line 18, when a `Person` object is instantiated. In the screen-shot (taken from Symantec Visual Cafe v2.5) below, line 19 is highlighted to indicate that it is the next statement to be executed.

In the box at the bottom labelled "Variables", observe the attributes of the variable `person`, the newly created Persistent Capable instance. Notice that `_vj_handle` is `null`. Recall that in Fundamental JVI, a `Handle` has a one-to-one relationship with a COD in the Object Cache. Since this `person` object has not been made persistent yet, there is no corresponding COD for it yet.

Since the object is not persistent, its `_vj_isUptodate` and `_vj_isDirty` fields are not relevant.


Visual Cafe - tutorial.VEP [Paused] - [c:\build_test\vji\tutorial\in\CreatePersonModified.java]
File Edit View Search Debug Insert Tools Variables Window Help
Workspace: Debug
Objects:
Events/Methods:
Name
Person.java
CreatePersonModified.
Imports
Line 20
Col

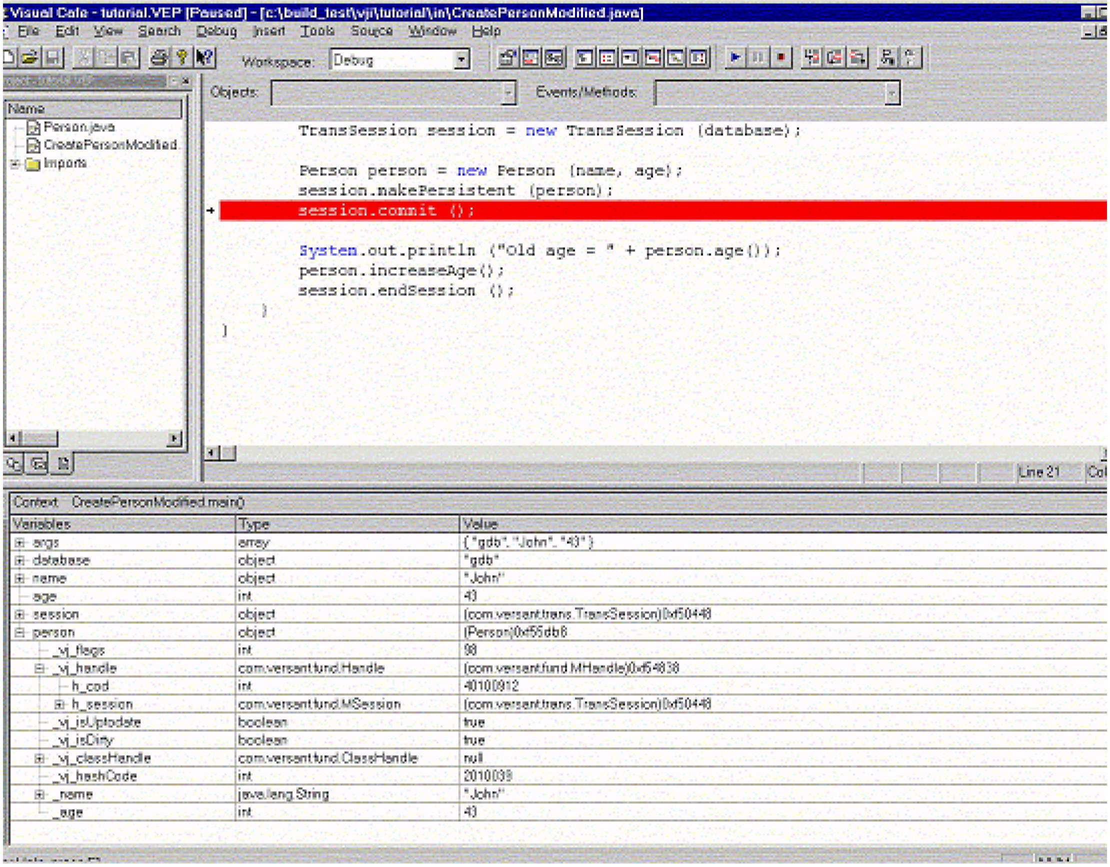
```
        TransSession session = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent (person);
        session.commit ();

        System.out.println ("Old age = " + person.age());
        person.increaseAge();
        session.endSession ();
    }
}
```

Context: CreatePersonModified.main()

Variables	Type	Value
args	array	{ "gdb", "John", "43" }
database	object	"gdb"
name	object	"John"
age	int	43
session	object	(com.versant.trans.TransSession)0xf50448
person	object	(Person)0xf55db8
_vj_flags	int	0
_vj_handle	com.versant.fund.Handle	null
no data available		
_vj_isUptodate	boolean	true
_vj_isDirty	boolean	true
_vj_classHandle	com.versant.fund.ClassHandle	null
_vj_hashCode	int	2010039
_name	java.lang.String	"John"
_age	int	43

Newly-Persistent State

After line 19 is executed and before line 20 is, the `Person` object has just been made persistent through the `makePersistent()` method, but it has not been committed yet.Unlike in the Transient State, the `_vj_Handle` field is set to a `Handle` object because a corresponding COD entry has been created in the COD table in the Object Cache. Once the object is made persistent it always has a `Handle`, unless the object is zapped. Also observe that the `_vj_isUptodate` flag and `_vj_isDirty` flag are both true, implying that this object does not need to be fetched from the database, but it has been modified. Therefore, it needs to be written to the database.



Visual Cafe - tutorial.VEP [Paused] - [c:\build_test\vji\tutorial\in\CreatePersonModified.java]

```
        TransSession session = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent (person);
        session.commit ();

        System.out.println ("Old age = " + person.age());
        person.increaseAge();
        session.endSession ();
    }
}
```

Context CreatePersonModified.main()

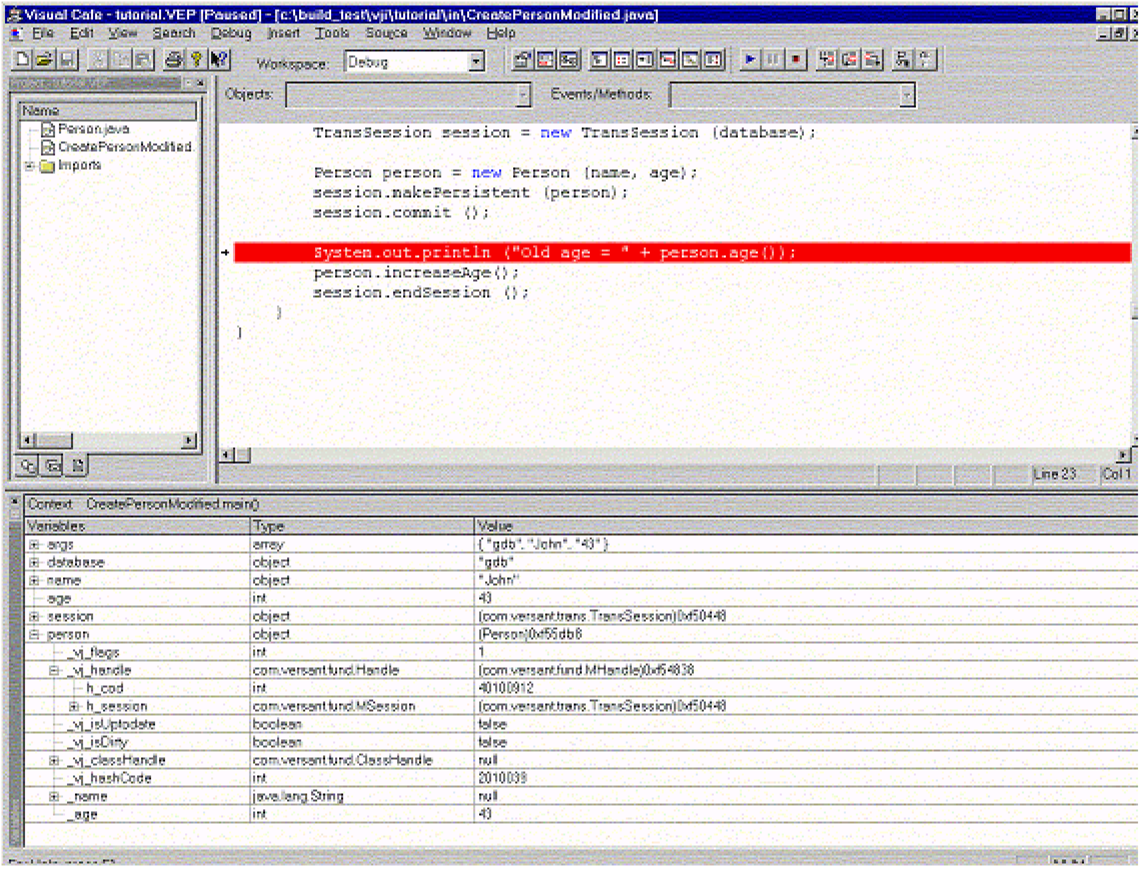
Variables	Type	Value
args	array	{ "gdb", "John", "43" }
database	object	"gdb"
name	object	"John"
age	int	43
session	object	(com.versant.trans.TransSession)0xf50448
person	object	(Person)0xf55db8
_vj_flags	int	98
_vj_handle	com.versant.fund.Handle	(com.versant.fund.MHandle)0xf54838
h_cod	int	40100912
h_session	com.versant.fund.MSession	(com.versant.trans.TransSession)0xf50448
_vj_isUptodate	boolean	true
_vj_isDirty	boolean	true
_vj_classHandle	com.versant.fund.ClassHandle	null
_vj_hashCode	int	2010039
_name	java.lang.String	"John"
_age	int	43

Committed State

At line 20 the transaction is committed. All the new and dirty objects are flushed to the database, objects are marked clean, and locks are released. A new transaction is started.

The `_vj_isUptodate` flag now becomes false. As the result of commit, the person object has been flushed to the database with no lock held on the object, other sessions on the same database can potentially modify this object. The next time this `person` object is accessed, JVI needs to fetch the object from the database into Java. Therefore, the up-to-date flag is set to false.

The `_vj_isDirty` flag is `false` because this person object is no longer dirty after commit.



Visual Cafe - tutorial.VEP [Paused] - [c:\build_test\vji\tutorial\in\CreatePersonModified.java]
File Edit View Search Debug Insert Tools Source Window Help
Workspace: Debug
Objects:
Events/Methods:
Name
Person.java
CreatePersonModified
Imports
Line 23
Col 1

```
        TransSession session = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent (person);
        session.commit ();

        System.out.println ("Old age = " + person.age());
        person.increaseAge();
        session.endSession ();
    }
}
```

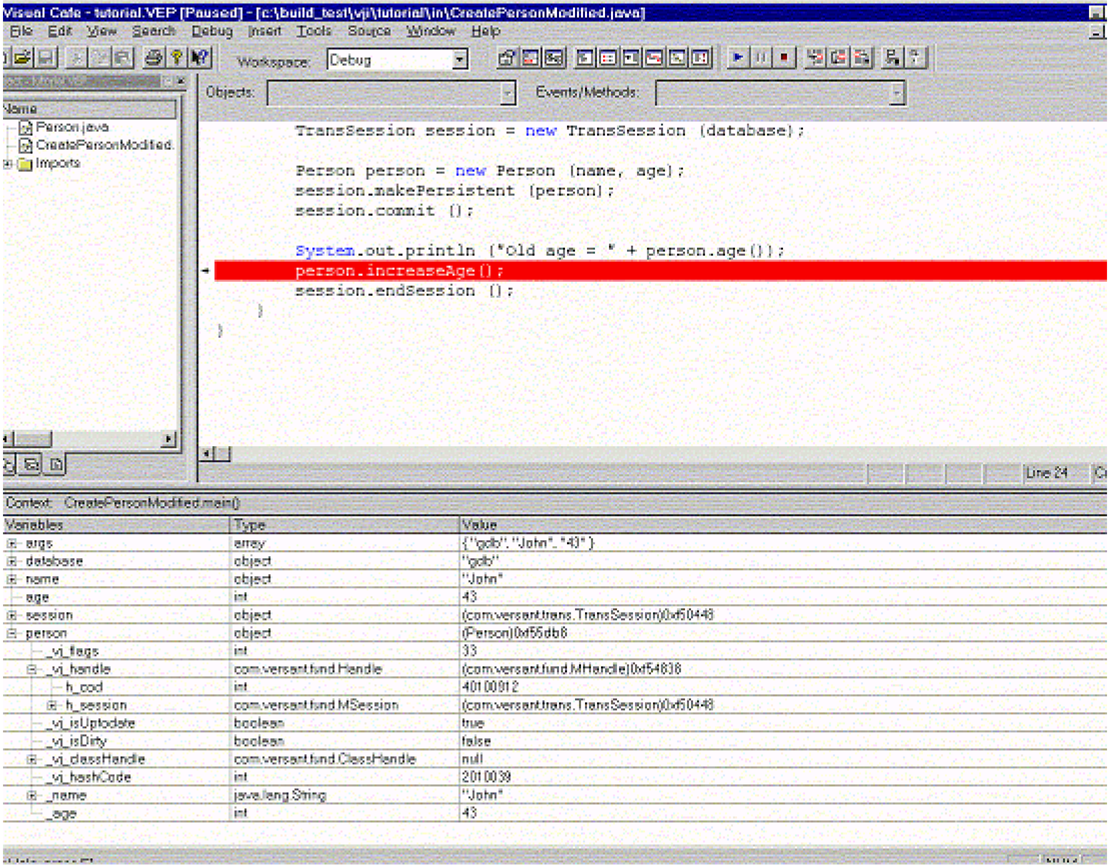
Context CreatePersonModified.main()

Variables	Type	Value
⊞ args	array	{ "gdb", "John", "43" }
⊞ database	object	"gdb"
⊞ name	object	"John"
age	int	43
⊞ session	object	(com.versant.trans.TransSession)0xf50448
⊟ person	object	(Person)0xf55db8
_vj_flags	int	1
⊟ _vj_handle	com.versant.fund.Handle	(com.versant.fund.MHandle)0xf54838
h_cod	int	40100912
⊞ h_session	com.versant.fund.MSession	(com.versant.trans.TransSession)0xf50448
_vj_isUptodate	boolean	false
_vj_isDirty	boolean	false
⊞ _vj_classHandle	com.versant.fund.ClassHandle	null
_vj_hashCode	int	2010039
⊞ _name	java.lang.String	null
_age	int	43

Clean State

At line 22, the person object is accessed by the age() method in the second transaction for the first time.

Since the object was marked out-of-date, JVI fetches the object from the database and places it into Java memory. The object is then flagged as up-to-date. Since no modification has been made to the object, it remains as clean.


Visual Cafe - tutorial.VEP [Paused] - [c:\build_test\vji\tutorial\in\CreatePersonModified.java]
File Edit View Search Debug Insert Tools Source Window Help
Workspace: Debug
Objects:
Events/Methods:
Name
Person.java
CreatePersonModified.
Imports
Line 24

```
        TransSession session = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent (person);
        session.commit ();

        System.out.println ("Old age = " + person.age());
        person.increaseAge();
        session.endSession ();
    }
}
```

Context: CreatePersonModified.main()

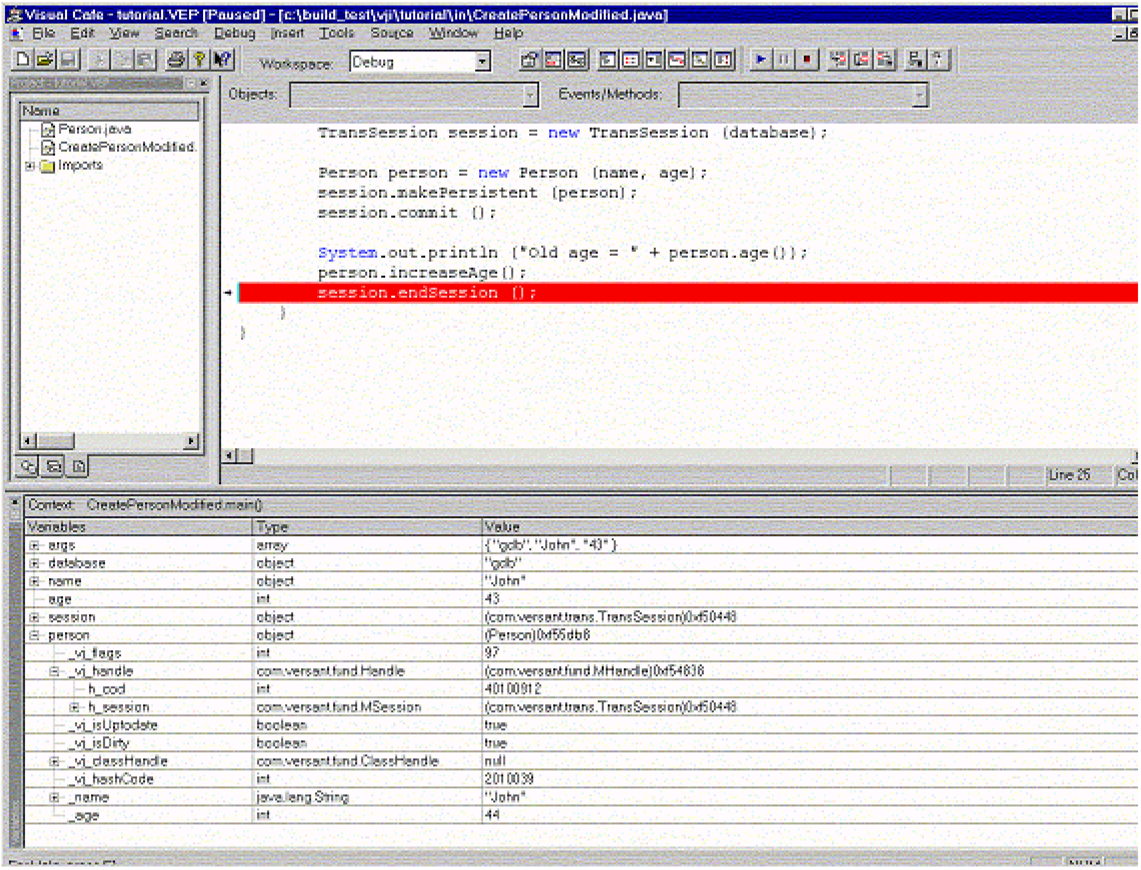
Variables	Type	Value
args	array	{"gdb", "John", "43" }
database	object	"gdb"
name	object	"John"
age	int	43
session	object	(com.versant.trans.TransSession)0xf50448
person	object	(Person)0xf55db8
_vj_flags	int	33
_vj_handle	com.versant.fund.Handle	(com.versant.fund.MHandle)0xf54838
h_cod	int	40100912
h_session	com.versant.fund.MSession	(com.versant.trans.TransSession)0xf50448
_vj_isUptodate	boolean	true
_vj_isDirty	boolean	false
_vj_classHandle	com.versant.fund.ClassHandle	null
_vj_hashCode	int	2010039
_name	java.lang.String	"John"
_age	int	43

Modified State

At line 23, the age instance variable of this person object is changed by the increaseAge() method.

In order to make the modification, JVI attempts to acquire a write lock on the object to have exclusive access to the object. The object is marked as dirty, so that the changes will be written when the transaction is committed.

The object remains up-to-date, since there is no need to re-fetch the object from the database when the object is accessed next.



Visual Cafe - tutorial.VEP [Paused] - [c:\build_test\vji\tutorial\in\CreatePersonModified.java]
File Edit View Search Debug Insert Tools Source Window Help
Workspace: Debug
Objects:
Events/Methods:
Name
Person.java
CreatePersonModified.
Imports
Line 26
Col

```
        TransSession session = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent (person);
        session.commit ();

        System.out.println ("Old age = " + person.age());
        person.increaseAge();
        session.endSession ();
    }
}
```

Context: CreatePersonModified.main()

Variables	Type	Value
args	array	{"gdb", "John", "43" }
database	object	"gdb"
name	object	"John"
age	int	43
session	object	(com.versant.trans.TransSession)0xf50448
person	object	(Person)0xf55db8
_vj_flags	int	97
_vj_handle	com.versant.fund.Handle	(com.versant.fund.MHandle)0xf54838
h_cod	int	40100912
h_session	com.versant.fund.MSession	(com.versant.trans.TransSession)0xf50448
_vj_isUptodate	boolean	true
_vj_isDirty	boolean	true
_vj_classHandle	com.versant.fund.ClassHandle	null
_vj_hashCode	int	2010039
_name	java.lang.String	"John"
_age	int	44

Summary

The following table summarizes the different states persistent objects go through in a typical application discussed above:

States	Transient	New or Dirty	Committed	Zapped
Up-to-date (`_vj_isUptodate`)	True	True	True	False
Dirty (`_vj_isDirty`)	True	True	False	False

Handle (`_vj_handle`)	null	MHandle	MHandle	MLoidHandle

TUTORIAL FOR TRANSPARENT JVI

Introduction

This tutorial contains a series of examples and explanations that demonstrate particular aspects of the Transparent binding.

We assume that you have installed the Versant database management system on your system and that you have some familiarity with basic database concepts, such as transactions, commit and rollback.

You will also need elemental knowledge of Java programming and object-oriented concepts, such as classes, inheritance, constructors, packages and garbage collection.

In the following, frequent references are made to files stored under your Java installation directory. Since this directory depends on the decisions you made at installation, the Java root directory for your installation is referred to as "`JVI`" in this tutorial. The directory that contains the tutorial files is referred to as "`TUT`" – this is the directory `tutorial/trans` in your installation.

Basic Procedures and Concepts

This part of the tutorial shows you the basics: how to make a Java object persistent and how to store these objects in the database.

Create a database: Before you can store objects in a database, the database must be created. This is done using the Versant `makedb` and `createdb` utilities.

Create a persistent class: This step proceeds just as in regular Java development: no special Java code is necessary to make a class persistent.

Create an application: Java applications are simply normal Java classes. JVI applications, however, are "database aware". For example, you must connect a "session" to a database before accessing persistent objects and then end this session when finished. The overhead involved for these steps are minimal.

Compile the Java classes: Java classes are compiled with the usual Java compiler. No special steps are necessary; however, care must be taken to correctly set the `CLASSPATH` environment variable needed by the Java compiler.

Enhance the Java classes: JVI is "transparent", which means that you must do very little work to allow objects to be persistently stored in a database. Instead, this work is done for you by a utility called the enhancer. The enhancer is a post-processor that modifies your class files so that they perform the extra steps necessary for persistence.

NOTE:- Please ensure that `CLASSPATH` is set properly before running the Enhancer.

Run the application: As in compilation, this step is the same as with normal Java development. Simply execute the application with the Java runtime environment. Again, it is necessary to properly set the `CLASSPATH`.

Create a Database

All of the examples in this tutorial require a database in which to store objects. Thus, our first step is to create this database. The name of the database is arbitrary; we will use the name `tutdb`. This can be easily changed, since all of the examples take the database name as a command-line parameter.

Creating a new Versant database is accomplished in two steps: first the database directory is made, and then the database files are created.

Before creating a new database, you must create a subdirectory for it under the Versant root database directory. To create the database directory for the database `tutdb`, run the makedb utility:

```
makedb tutdb
```

This creates a subdirectory, owned by you, under the Versant root database directory.

To see the location of this root directory, use the `oscp` utility:

```
oscp -d
```

In addition, the `makedb` utility creates front end and backend profiles; we will not be using these profiles in the tutorial.

For more information on database profiles, please refer to Chapter “Database Profiles” in *Versant Database Administration Manual*.

Create the Database

To create the database, use the `createdb` utility:

```
createdb tutdb
```

This creates the database system storage and log files in the database directory `tutdb`.

Create a Persistent Class

Create a Java Class

To create a Java class for persistent objects, first write the `.java` source code file as usual. No changes are necessary to allow objects of this class to be stored in the database. These changes are done automatically for you in the enhancement process.

Our first examples will use a `Person` class, which is included in the tutorial directory. To indicate that instances of `Person` are persistent objects, a corresponding line will be added to the configuration file that is read during the enhancement process. This will be described in greater detail in the section on enhancement.

TUT/in/Person.java

```
public class Person
{
    String name;
    int age;

    Person (String aName, int anAge)
    {
        name = aName;
        age  = anAge;
    }

    public String toString ()
    {
        return "Person: " + name + " age: " + age;
    }
}
```

NOTE:- In the `Person` class above the `name` and `age` attributes have not been hidden with the private or protected access modifiers. Although not allowing publicly accessible fields is often considered standard object-oriented programming practice, this would only serve to complicate the example and distract from the main concepts being illustrated.

Create an Application

Our first example application will simply create instances of the `Person` class. The sample application makes a persistent object using the following process:

- Connect to the database and start a transaction by creating a new `TransSession` object.
- Create a new instance of class `Person` as usual, using operator `new`.
- Commit the transaction by ending the session.

TUT/in/CreatePerson.java

```
import com.versant.trans.*;

public class CreatePerson
{
    public static void main (String[] args)
    {
        if (args.length != 3) {
            System.out.println
                ("Usage: java CreatePerson <database> <name> <age>");
            System.exit (1);
        }
        String       database = args [0];
        String       name     = args [1];
        int          age      = Integer.parseInt (args[2]);
        TransSession session  = new TransSession (database);

        Person person = new Person (name, age);
        session.makePersistent (person);

        session.endSession ();
    }
}
```

Import JVI Classes

In the example application, the JVI class `TransSession` is used. To import this class into the namespace of the application program, use the Java import directive. This class is located in the JVI package `com.versant.trans`.

Start Database Session

Before any persistent objects can be created or accessed, you must connect to the database by starting a database session. Starting a database session initiates Versant processes that allow you to access Versant databases. You can use Versant database methods and persistent objects only within a session.

In the Transparent binding, starting a database session is achieved by creating a new instance of the `TransSession` class. The constructor for this class has two parameters: a set of properties and a session capability.

Beginning a session can be controlled by several options. These options are set in the `Properties` object that is given to the `TransSession` constructor. For now, we are only interested in one option: the name of the database.

For more information, on the available options when starting a session, please refer to Section on "`TransSession` class" in the *Java Versant Interface Reference Manual.*

Start a Transaction

Starting a database session also starts a transaction. All changes to persistent objects are written to the database only if the transaction successfully commits.

Create a Persistent Object

Since the `Person` class will be designated as a persistent class in the configuration file, no special code must be written to create persistent `Person` objects. Simply invoke the `new` operator as usual. The `TransSession.makePersistent` method causes the given object to be stored persistently in the database.

Commit the Transaction

Ending the session causes the active transaction to commit. To commit a transaction without ending the session, use the `TransSession.commit` method.

Compile the Java Classes

To compile the classes of a JVI application, simply invoke the Java compiler as usual. No special action needs to be taken. However, it is necessary to correctly set the environment variable `CLASSPATH`, so that the JVI classes can be accessed by the compiler.

The JVI classes are located in a `.jar` file in the `<VERSANT_ROOT>/lib` release directory. The name of this file depends on both the JVI and Java version numbers; for example, the `.jar` file for JVI

7.0.1 with JDK 1.4 is named `jvi7.0.1-jdk1.4.jar`. Add the `.jar` file to your `CLASSPATH` environment variable. In addition, it is necessary to add the directory containing the Java source files to your `CLASSPATH`. This can be done by including the "current" directory, represented by a single period.

The exact method for setting `CLASSPATH` depends on the system being used. Please refer to your Java documentation for correctly setting `CLASSPATH`.

To compile the two Java classes:

```
cd TUT/in/
javac Person.java CreatePerson.java
```

Enhance the Java Classes

Enhancement is where the "magic" of the JVI Transparent Package takes place. The enhancer is a post-processing utility that modifies your Java classes so that instances can be stored in the database. It also augments code that accesses these persistent objects so that they correctly interact with the database.

The enhancer is termed a "postprocessor" because it manipulates Java `.class` files, not `.java` source files. That is, the input to the enhancer is a set of compiled Java classes, because enhancement occurs after compilation. The output of the enhancer is a corresponding set of class files; but these classes have been modified to correctly work with Versant databases.

Post-processing is feasible for two main reasons:

- Java `.class` files contain virtual machine language instructions that are independent of any one processor. Thus, the same `.class` files can be executed on different platforms. This means that the enhancer does not have to consider platform-specific details.
- The `.class` files have a relatively simple structure that make them easy to read and modify in utilities such as the enhancer.

Although it actually does quite a bit more, the enhancer performs three main functions on its input `.class` files:

- The schema of the persistent classes is captured, and code is generated to inform the database of this schema.
- Code is generated that moves objects from the database into Java memory and vice versa.
- Application code that accesses and modifies persistent objects is augmented to mark objects as dirty and fetch objects from the database, so that changes to these objects are written to the database when the transaction is committed.

A configuration file, usually named `config.vj`, controls the behavior of the enhancer. This file specifies the persistence category of each of the classes that are being enhanced. The configuration file for the example application above would look as follows:

```
c Person
a CreatePerson
```

The letter "c" indicates that instances of the `Person` class are categorized as Persistent Capable. This means that `Person` objects can be stored in the Versant database and the enhancer will modify the `Person` class so that persistence is possible.

The `CreatePerson` class, however, is only Persistent Aware (category "a"). This means that `CreatePerson` instances will not be normally stored in the database, but the methods of class `CreatePerson` will be augmented to correctly work with other persistent objects, such as `Person` objects.

The enhancer looks for `.class` files in an input directory and deposits modified `.class` files in an output directory. These two directories are specified on the command line when the enhancer is executed.

The input directory is the top of the package hierarchy for all of the classes comprising the JVI application. After the enhancer is run, the output directory mimics the structure of the input directory.

The input and output directories for the tutorial examples can be found in your JVI installation as the `TUT/in/` and `TUT/out/` directories.

The enhancer is itself a Java application. Then, assuming the `CLASSPATH` is set as described above in the section on compilation and that you have a subdirectory named "`out`", run the enhancer:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

The `-verbose` flag causes the enhancer to indicate the persistence category of each class, and to display status information as the enhancement process proceeds.

Afterwards, the `out` directory will contain the two enhanced `.class` files, as well as an additional file : `Person_Pickler_Vj.class`. This class assists in making the `Person` objects persistent.

Run the Application

To run the sample application, simply invoke the Java interpreter as usual. However, it is very important to include the directory containing the enhanced `.class` files in the `CLASSPATH`. Failure to do so will result in run-time errors. Since your `CLASSPATH` contains the current directory (represented by "."), this can be accomplished simply using the command:

```
cd TUT/out/
```

In addition, since the JVI libraries are implemented using native methods (functions written in the C language), the operating system must be able to locate the JVI dynamic-link libraries. These are located in the `<VERSANT_ROOT>/lib/` subdirectory of your JVI installation. On Unix machines, this directory is normally added to your `LD_LIBRARY_PATH` environment variable. On Windows machines, this directory must be added to your `PATH`.

To execute the `CreatePerson` application:

```
java CreatePerson tutdb Bob 28
```

The above command adds a single `Person` object to the `tutdb` database, with name Bob and age 28. This can be seen using the `db2tty` utility:

```
db2tty -d tutdb -i Person
```

For more information, on the `db2tty` utility, please refer to the *Versant Database Administration Manual*.

Accessing Objects

Now that you have seen the basics, including how to put objects into a database, let us look at how to access those objects.

There are two basic ways of finding existing objects in the database: with roots or with queries. Once a persistent object has been found by either of these two methods, its fields can be read and modified and its methods can be invoked. Any changes made will be written back to the database at the next transaction commit.

Finding Objects with Roots

A "root" is a persistent object that has been given a name. This name can be used to find the object later. A root name is a bit like a file name, although the system of roots in JVI is much simpler than a true file system. In particular, there is one "space" for root names in each database.

Root names should be applied to only a relatively small number of objects in a database. Many database applications have complex, connected "graphs of objects, a root provides a simple starting point for these graphs.

There are three fundamental root operations:

- Making a new root by giving an object a name.
- Finding an object with a given root name.
- Deleting a root name.

First let's make a new root by giving a name to an object. The following example application, `CreatePersonWithRoot`, creates a new `Person` object and gives it a root name.

TUT/in/CreatePersonWithRoot.java

```
import com.versant.trans.*;
public class CreatePersonWithRoot
{
    public static void main (String[] args)
    {
        if (args.length != 4) {
            System.out.println ("Usage: java CreatePersonWithRoot" +
              "<database> <name> <age> <root>");
            System.exit (1);
        }
        String       database = args [0];
        String       name     = args [1];
        int          age      = Integer.parseInt (args[2]);
        String       root     = args [3];
        TransSession session  = new TransSession (database);
        Person person = new Person (name, age);
        session.makeRoot (root, person);
        session.endSession ();
    }
}
```

This application is exactly like the previous `CreatePerson` program, except that it calls the `TransSession.makeRoot` method instead of `makePersistent`. This allows us to give the object a root name, which is taken from the command line.

To run this application, go through the same steps as described in the previous part of the tutorial.

Compile the `.java` source file:

```
cd TUT/in/
javac CreatePersonWithRoot.java
```

Modify the configuration file:

Add the necessary line to the configuration file, `TUT/config.vj`, to inform the enhancer that the `CreatePersonWithRoot` class is Persistent Aware. This line would read:

```
a CreatePersonWithRoot
```

NOTE:- The configuration file supplied with the JVI installation has already been updated to work with this and the following examples.

Enhance the `.class` file and put output in subdirectory `out`:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java CreatePersonWithRoot tutdb Mary 43 mary_root
```

This creates a persistent `Person` object identified by the root name `mary_root`. This root name can be used to subsequently retrieve the object, using the `TransSession.findRoot` method. The following program illustrates the use of `findRoot`:

TUT/in/FindPersonWithRoot.java

```
import com.versant.trans.*;
public class FindPersonWithRoot
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java FindPersonWithRoot <database> <root>");
            System.exit (1);
        }
        String       database = args [0];
        String       root     = args [1];
        TransSession session  = new TransSession (database);

        Person person = (Person) session.findRoot (root);
        System.out.println ("Found " + person);

        session.endSession ();
    }
}
```

The findRoot method returns an instance of Object, so a typecast must be used to recover the actual type, Person. If no root has the given root name, then findRoot throws an exception.

Now compile, enhance, and run the FindPersonWithRoot application.

Compile the .java source file:

```
cd TUT/in/
javac FindPersonWithRoot.java
```

Update the configuration file TUT/config.vj:

```
a FindPersonWithRoot
```

Enhance the .class file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java FindPersonWithRoot tutdb mary_root
```

Running the application gives the output:

```
Found Person: Mary age: 43
```

Finding Objects with Queries

Versant provides a query language, VQL, to search for persistent objects that match certain criteria. JVI supports simple VQL queries. These queries can be used to find objects that have been stored in the database.

For more information refer to “VQL Queries” on page 57, in "Chapter 2 - Fundamental JVI".

The following program finds all Person objects located in a database.

```
TUT/in/FindPersonWithVQL.java
import com.versant.trans.*;
import java.util.*;
public class FindPersonWithVQL
{
    public static void main (String[] args)
    {
        if (args.length != 1) {
            System.out.println
                ("Usage: java FindPersonWithVQL <database>");
            System.exit (1);
        }
        String       database = args [0];
        TransSession session  = new TransSession (database);

        VQLQuery query =
            new VQLQuery (session, "select selfoid from Person");
        Enumeration e = query.execute ();

        if ( !e.hasMoreElements() ) {
            System.out.println ("No Person objects were found.");
        } else {
            while ( e.hasMoreElements() ) {
                Person person = (Person) e.nextElement ();
                System.out.println ("Found " + person);
            }
        }

        session.endSession ();
    }
}
```

Finding all `Person` objects proceeds in three steps:

4. First, the query object is constructed, with a query that means, "find all objects from the `Person` class".
5. Next, the query is executed. This tells the Versant database to find matching objects.
6. Finally, the matching objects are fetched from the database using an instance of the Java interface `java.util.Enumeration`. The enumeration provides restricted access to a sequence of objects, so that the objects are not retrieved from their database until explicitly demanded by the application program with the `nextElement` method.

Compile, enhance, and run this application using steps similar to those shown previously.

Compile the `.java` source file:

```
cd TUT/in/
javac FindPersonWithVQL.java
```

Update the configuration file `TUT/config.vj`:

```
a FindPersonWithVQL
```

Enhance the `.class` file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java FindPersonWithVQL tutdb
```

Running the application gives the following output:

```
Found Person: Bob age: 28
Found Person: Mary age: 43
```

Changing Persistent Objects

This example shows how to modify database persistent objects.

Like the previous program, it finds all `Person` objects in a database. This time, however, instead of simply displaying the contents of the object, the age of each `Person` object is increased by one - after all, we are all getting older every year!

TUT/in/IncreaseAge.java

```
import com.versant.trans.*;
import java.util.*;

public class IncreaseAge
{
    public static void main (String[] args)
    {
        if (args.length != 1) {
            System.out.println
                ("Usage: java IncreaseAge <database>");
            System.exit (1);
        }
        String       database = args [0];
        TransSession session  = new TransSession (database);

        VQLQuery query = new VQLQuery
            (session, "select selfoid from Person");
        Enumeration e = query.execute ();

        if ( !e.hasMoreElements() ) {
            System.out.println ("No Person objects were found.");
        } else {
            while ( e.hasMoreElements() ) {
                Person person = (Person) e.nextElement ();
                person.age++;
                System.out.println ("Increasing " + person.name +
                                    "'s age to " + person.age);
            }
        }
        session.endSession ();
    }
}
```

Compile the `.java` source file:

```
cd TUT/in/
javac IncreaseAge.java
```

Update the configuration file `TUT/config.vj`:

```
a IncreaseAge
```

Enhance the `.class` file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java IncreaseAge tutdb
```

This shows the following output:

```
Increasing Bob's age to 29
Increasing Mary's age to 44
```

You can use the `FindPersonWithVQL` program to verify that the ages have indeed increased.

Deleting Persistent Objects

Persistent objects in a Versant database remain in the database until explicitly deleted. To delete objects from a database, use the `TransSession.deleteObject` method.

NOTE:-

- If the server profile parameter `commit_delete` is `OFF`, this function will send the delete request to the source database and the object is deleted immediately.
- If `commit_delete` is `ON`, this function will acquire a Write Lock on the object and set its status as "`Marked for deletion`". The object will be physically deleted at commit.

  If the transaction commits, the objects are physically deleted. If a rollback occurs, these objects are un-marked and their status restored.
- Queries run on the database will not include objects marked for deletion in the result sets.

The following program will delete an object with a given root name.

TUT/in/DeletePersonWithRoot.java

```
import com.versant.trans.*;
public class DeletePersonWithRoot
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java DeletePersonWithRoot <database> <root>");
            System.exit (1);
        }

        String database = args [0];
        String root     = args [1];

        TransSession session = new TransSession (database);

        Person person = (Person) session.findRoot (root);
        session.deleteObject (person);

        session.endSession ();
    }
}
```

Unlike C++, Java does not support a delete operation for dynamically allocated objects. Instead, Java relies on garbage collection to rid memory of unreferenced objects. Therefore, the `deleteObject` method deletes the persistent object from the database only — not from memory! After deletion, the object in memory should not be accessed. The object will be deleted from the database at transaction commit.

Deleting an object is not the same as deleting a root: deleting a root simply removes the root name associated with the object and does not delete the object from its database.

Now compile, enhance and run this program.

Compile the `.java` source file:

```
cd TUT/in/
javac DeletePersonWithRoot.java
```

Update the configuration file `TUT/config.vj`:

```
a DeletePersonWithRoot
```

Enhance the `.class` file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java DeletePersonWithRoot tutdb mary_root
```

Running the same program, a second time generates an exception because the object was deleted from the database and the root name removed.

Using Links

In a Versant database, a "link" is a reference within a persistent object to another persistent object. A link is essentially an attribute that contains the LOID of a persistent object. A Versant link is analogous to a pointer or reference in a programming language. In fact, the analogy is so strong that links are implemented as normal Java references in the JVI Transparent Package.

Links are extremely easy to use. You simply define your classes in exactly the same way that you would when writing a normal, non-database application. The enhancer takes care of all of the work of converting references to links and fetching objects from the database when they are accessed.

To illustrate how links work in a JVI Transparent application, we will use simple `Employee` and `Department` classes. The `Employee` class contains a reference to the `Department` class: each employee belongs to one department. Similarly, the `Department` class contains a reference to the `Employee` class: one employee manages each department.

TUT/in/Employee.java

```
public class Employee extends Person
{
    Department department;
    double salary;

    Employee (String aName, int anAge, double aSalary)
    {
        super (aName, anAge);
        salary = aSalary;
    }

    public String toString ()
    {
        return "Employee: " + name + " age: " + age +
               " salary: " + salary + " " + department;
    }
}
```

TUT/in/Department.java

```
public class Department
{
    String name;
    Employee manager;

    Department (String aName, Employee aManager)
    {
        name    = aName;
        manager = aManager;
    }

    public String toString ()
    {
        return "Department: " + name + " manager: " +
            (manager == null ? "nobody" : manager.name);
    }
}
```

The following application creates some `Employee` objects and stores them in the database.

```
TUT/in/AddEmployees.java

import com.versant.trans.*;
public class AddEmployees
{
    public static void main (String[] args)
    {
        if (args.length != 1) {
            System.out.println
                ("Usage: java AddEmployees <database>");
            System.exit (1);
        }
        String       database = args [0];
        TransSession session  = new TransSession (database);

        Employee the_boss   = new Employee ("The Boss", 42, 110000);
        Employee jane_jones = new Employee ("Jane Jones", 24, 80000);
        Employee john_doe   = new Employee ("John Doe", 25, 75000);
        Employee lois_line  = new Employee ("Lois Line", 36, 70000);

        Department engineering =
            new Department ("Engineering", the_boss);
        Department marketing =
            new Department ("Marketing",   lois_line);

        the_boss  .department = engineering;
        jane_jones.department = engineering;
        john_doe  .department = marketing;
        lois_line .department = marketing;

        session.makePersistent (the_boss);
        session.makePersistent (jane_jones);
        session.makePersistent (john_doe);
        session.makePersistent (lois_line);
        session.makePersistent (engineering);
        session.makePersistent (marketing);

        session.endSession ();
    }
}
```

As before, compile, enhance, and run this program.

Compile the `.java` source files:

```
cd TUT/in/
javac Employee.java Department.java AddEmployees.java
```

Update the configuration file `TUT/config.vj`:

```
p Employee
p Department
a DeletePersonWithRoot
```

Enhance the `.class` files:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java AddEmployees tutdb
```

Running this application will add four `Employee` objects to the database.

To see that the four objects exist, you can use the `db2tty` utility:

```
db2tty -d tutdb -i Employee
```

However, the `FindPersonWithVQL` application will also find these `Employee` objects:

```
cd TUT/out/
java FindPersonWithVQL tutdb
```

This displays the following output:

```
Found Person: Bob age: 29
Found Employee: The Boss age: 42 salary: 11000.0 Department: Engin
eering manager: The Boss
Found Employee: Jane Jones age: 24 salary: 80000.0 Department:
```

```
Engineering manager: The Boss
Found Employee: John Doe age: 25 salary: 75000.0 Department: Market-
 manager: Lois Line
Found Employee: Lois Line age: 36 salary: 70000.0 Department: Market-
 manager: Lois Line
```

Since the `Employee` class extends `Person`, these `Employee` objects are also `Person` objects. This is why the VQL query above matched the `Employee` objects as well. (You can tell VQL not to match sub-types by using the keyword `only`.) Similarly, the `IncreaseAge` program will modify the ages of all `Person` objects, including the `Employee` objects.

To run the `IncreaseAge` program:

```
cd TUT/out/
java IncreaseAge tutdb
```

This displays the following messages:

```
Increasing Bob's age to 30
Increasing The Boss's age to 43
Increasing Jane Jones's age to 25
Increasing John Doe's age to 26
Increasing Lois Line's age to 37
```

Transitive Persistence

So far you have seen just one way of storing objects persistently in the Versant database: explicitly calling the makePersistent or makeRoot methods. However, a Persistent Capable object can also become persistent through a mechanism called transitive persistence. If a Persistent Capable object is referenced in a persistent object, it too becomes persistent.

To illustrate the concepts of Persistent Capable classes and transitive persistence, we will use the following simple `LinkedList` class.

TUT/in/LinkedList.java

```
public class LinkedList
{
    int label;
    LinkedList next_node;

    LinkedList (int aLabel, LinkedList list)
    {
        label     = aLabel;
        next_node = list;
    }

    public String toString ()
    {
        return label + (next_node == null ? "" : " " + next_node);
    }
}
```

The `LinkedList` class will be Persistent Capable, so the appropriate line should be added to the configuration file:

```
c LinkedList
```

Now consider the following application, which creates a linked list.

```
TUT/in/CreateLinkedList.java

import com.versant.trans.*;

public class CreateLinkedList
{
    public static void main (String[] args)
    {
        if (args.length != 1 && args.length != 2) {
            System.out.println
                ("Usage: java CreateLinkedList <database> [root]");
            System.exit (1);
        }
        String       database = args [0];
        String       root     = args.length == 2 ? args [1] : null;
        TransSession session  = new TransSession (database);

        LinkedList list = null;
        for (int i = 0; i < 5; i++) {
            list = new LinkedList (i, list);
        }
        if (root != null)
            session.makeRoot (root, list);

        session.endSession ();
    }
}
```

This application creates, in memory, a linked list with five nodes, labelled from 4 down to 0. The program takes an optional second command-line argument, the root name. If the root name is given, then the head of the linked list is made persistent by the makeRoot method. On the other hand, if the optional argument is not given, then none of the linked list nodes are made persistent.

Now compile, enhance and run this application.

Compile the .java source file:

```
cd TUT/in/
javac CreateLinkedList.java
```

Update the configuration file TUT/config.vj:

```
c LinkedList
a CreateLinkedList
```

Enhance the .class files:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application, first without giving the optional argument:

```
cd TUT/out/
java CreateLinkedList tutdb
```

By using the db2tty utility, you can see that there are no LinkedList objects in the database:

```
db2tty -d tutdb -i LinkedList
```

Now run the same application, this time with a root name:

```
java CreateLinkedList tutdb list_root
```

Now the db2tty utility will show the five LinkedList nodes in the database. You can also see them with the following application:

```
TUT/in/FindLinkedList.java

import com.versant.trans.*;

public class FindLinkedList
```

```
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java FindLinkedList <database> <root>");
            System.exit (1);
        }
        String       database = args [0];
        String       root     = args [1];
        TransSession session  = new TransSession (database);

        LinkedList list = (LinkedList) session.findRoot (root);
        System.out.println ("Found List: " + list);
        session.endSession ();
    }
}
```

Now compile, enhance, and run this program with the same root name as before.

Compile the `.java` source file:

```
cd TUT/in/
javac FindLinkedList.java
```

Update the configuration file `TUT/config.vj`:

```
a FindLinkedList
```

Enhance the `.class` file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java FindLinkedList tutdb list_root
```

This shows the following output:

```
Found List: 4 3 2 1 0
```

What does this mean? It means that even though just a single object was explicitly made persistent (by the `makeRoot` method), five objects were actually persistently stored in the database. The reason is transitive persistence. Since the head of the linked list, node 4, is persistent and contains a link to node 3, node 3 is transitively persistent as well. In addition, since node 3 is persistent and contains a link to node 2, node 2 is persistent, and so on.

While this example demonstrated a very simple form of transitive persistence, where the linked objects were arranged in a regular, "linear" form, transitive persistence does apply too much more complicated linked structures as well. Any Persistent Capable object becomes persistent if it can be reached from a persistent object by following links.

Collections

Arrays

In JVI, a collection is a group of persistent objects. JVI supports many different kinds of collections, but the simplest is the array. You can use regular Java arrays in your database application without change.

Collections can be used as fields in a persistent class. When used in this way, collections can help implement one-to-many or many-to-many relationships. As a simple (and incomplete) example, consider the following `Student` and `Course` classes. Each `Student` object has an array of courses, and each `Course` object has an array of students. When a student adds a course, both of these arrays are updated. This maintains the many-to-many relationship between the two classes.

```
Student.java

public class Student extends Person
{
   int id_number;
   Course[] courses;
   int num_courses;

   void addCourse (Course aCourse)
   {
      courses [num_courses] = aCourse;
      num_courses++;
      aCourse.students [aCourse.num_students] = this;
      aCourse.num_students++;
   }
}
```

```
Course.java

public class Course
{
   String course_code;
   String description;
   Student[] students;
   int num_students;
   Person instructor;
}
```

Since this is an incomplete example, there is no sample application to build and run.

Vectors

Another kind of collection is the vector. A vector is similar to an array, but does not have a fixed size and can grow to meet the needs placed upon it. You are probably familiar with the `Vector` class that is part of the standard Java library. The `VVector` class is a persistent replacement for the `Vector` class. A `VVector` object can itself be placed in the database and can contain persistent objects.

As a simple example of using the `VVector` class, consider the following program. It allows new `Person` objects to be added to named root `VVector` objects.

The `VVector` class is located in the `com.versant.util` package; the example below has the appropriate import statement.

```
TUT/in/CreatePersonVector.java

import com.versant.fund.*;
import com.versant.trans.*;
import com.versant.util.*;

public class CreatePersonVector
{
    public static void main (String[] args)
    {
        if (args.length != 4) {
            System.out.println ("Usage: java CreatePersonVector" +
                                " <database> <name> <age> <root>");
            System.exit (1);
        }
        String       database = args [0];
        String       name     = args [1];
        int          age      = Integer.parseInt (args[2]);
        String       root     = args [3];
        TransSession session  = new TransSession (database);

        VVector vector;
        try {
            vector = (VVector) session.findRoot (root);
        } catch (VException vex) {
            int errno = vex.getErrno();
            if (errno == Constants.EVJ_ROOTNAME_DOES_NOT_EXIST) {
                vector = new VVector ();
                session.makeRoot (root, vector);
            } else {
                throw vex;
            }
        }

        Person person = new Person (name, age);
        vector.addElement (person);

        session.endSession ();
    }
}
```

The application proceeds in three basic steps.

1. The root name that is given as a command-line argument is used to find a `VVector` object. If no object exists that has a matching root name, a new `VVector` object is created.
2. A new `Person` object is created with the name and age given as command-line arguments.
3. The newly created `Person` object is added to the `VVector`.

Now compile, enhance, and run this program, creating several `Person` objects contained in two `VVector` objects.

Compile the `.java` source file:

```
cd TUT/in/
javac CreatePersonVector.java
```

Update the configuration file `TUT/config.vj`:

```
a CreatePersonVector
```

Enhance the `.class` file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application several times:

```
cd TUT/out
java CreatePersonVector tutdb Alex 24 friends
java CreatePersonVector tutdb Sarah 21 friends
java CreatePersonVector tutdb Lee 36 friends
java CreatePersonVector tutdb Casper 256 enemies
java CreatePersonVector tutdb Garfield 14 enemies
```

To see these collections, use the following application:

```
TUT/in/FindPersonVector.java

import com.versant.trans.*;
import com.versant.util.*;

public class FindPersonVector
{
    public static void main (String[] args)
    {
        if (args.length != 2) {
            System.out.println
                ("Usage: java FindPersonVector <database> <root>");
            System.exit (1);
        }
        String       database = args [0];
        String       root     = args [1];
        TransSession session  = new TransSession (database);

        VVector vector = (VVector) session.findRoot (root);
        for (int i = 0; i < vector.size(); i++) {
            Person person = (Person) vector.elementAt (i);
            System.out.println ("Found " + person);
        }
        session.endSession ();
    }
}
```

This program locates the VVector with the given root name and iterates through the collection, printing each of its elements. Of course, iteration could have been done with an Enumeration as well, using the elements method on the vector.

Compile, enhance, and run the application.

Compile the .java source file:

```
cd TUT/in/
javac FindPersonVector.java
```

Update the configuration file TUT/config.vj:

```
a FindPersonVector
```

Enhance the .class file:

```
java com.versant.Enhance –config ../config.vj -out ../out -verbose .
```

Run the application:

```
cd TUT/out/
java FindPersonVector tutdb friends
```

This displays the list of "friends":

```
Found Person: Alex age: 24
Found Person: Sarah age: 21
Found Person: Lee age: 36
```

Note that this application would work just as well using the java.util.Vector class instead of VVector. However, VVector is more efficient when the collection contains a large number of persistent objects.

As another example of the use of vectors, we can write the previous Student and Course classes using VVector instead. This way, there is no limit to the number of courses a student can add or the number of students a course can accept. In addition, the vector keeps track of the number of elements stored in it. This simplifies the application code.

Student.java

```
import com.versant.util.*;

public class Student extends Person
{
   int id_number;
   VVector courses;

   void addCourse (Course aCourse)
{
      courses.addElement (aCourse);
      aCourse.students.addElement (this);
   }
}

Course.java
import com.versant.util.*;

public class Course {
   String course_code;
   String description;
   VVector students;
   Person instructor;
}
```

Other Collection Classes

JVI includes several collection classes other than `VVector`. These include `VHashtable` in the `com.versant.util` package, which is a replacement for the standard Java `Hashtable` class, as well as generic data structures such as `Bag`, `Set` and `List` in the `com.versant.odmg` package.

Second Class Objects

All of the objects that you have seen in this tutorial are First Class Objects. This means that the Versant database recognizes them as individual objects, and has assigned a LOID to each one that is stored in the database.

However, JVI supports another kind of persistent object, the Second Class Object. A Second Class Object cannot exist in the database in its own right; it must exist as an attribute of a First Class Object. The Second Class Object is in some sense "subordinate" to a First Class Object.

This First Class Object is sometimes referred to as the "owner" of the Second Class Object. Each Second Class Object can have only one owner.

Second Class Objects use the Java serialization mechanism to achieve persistent storage. Serialization turns an object into a byte stream — that is, an array or sequence of bytes that can be de-serialized to reconstitute the object.

There is a tradeoff involved in using First Class or Second Class Objects. First Class Objects require more database overhead, since the database has to handle each First Class Object separately. Second Class Objects, on the other hand, do not work easily with database queries (since the database does not "understand" the Java serialization format) and involve some extra runtime serialization overhead. In addition, Second Class Objects cannot be shared between First Class Objects, because each Second Class Object can have only one owner. Fortunately, other than the restriction on sharing Second Class Objects, you do not have to change the way in which you write your Java programs to measure this tradeoff; only the configuration file need be changed.

As a simple example of using Second Class Objects, consider the following `Friend` and `Address` classes. Each of your friends has an address, but you might not want these addresses to occupy their own object in the database, because the `Address` class is really just a way of grouping the related information so that it can be dealt with at once in your application program.

To indicate that the instances of the `Address` class should be Second Class Objects, mark this class as category "d" in the configuration file:

```
c Friend
d Address
Friend.java
public class Friend extends Person
{
   String phone_number;
   Address address;
}

Address.java
public class Address
{
   String street;
   String city;
   String state;
   int zip_code;
}
```

Now, whenever a `Friend` object is written to the database, the `Address` object will be serialized along with it. If you change the category from "d" to "p" (Always Persistent) or "c" (Persistent Capable), your application will behave in the same way; only the representation in the database will change.

CHAPTER 4

ODMG JVI

This Chapter gives detailed explaination about the concepts and usage of “ODMG JVI”.

The Chapter covers the following in detail:

- Introduction
- Database Class
- Transaction Class
- Collection Classes
- Exception Classes
- Queries

INTRODUCTION

Package Contents

The ODMG JVI allows you to use the standard interface methods and data constructs specified in the Object Database Management Group 2.0 Java binding specifications.

The `com.versant.odmg` package contains the following interfaces and classes:

Basic Classes	**Transaction Database**
JVI specific	`*ODMGSession`
Collection Interfaces	`Bag`
	`Collection`
	`List`
	`Set`
	`Map`
Exception Classes	`DatabaseClosedException`
	`DatabaseNotFoundException`
	`DatabaseOpenException`
	`DatabaseReadOnlyException`
	`LockNotGrantedException`
	`ObjectNameNotFoundException`
	`ObjectNameNotUniqueException`
	`ODMGException`
	`ODMGRuntimeException`
	`TransactionAbortedException`
	`TransactionInProgressException`
	`TransactionNotInProgressException`

Exception Classes*	`DatabaseClosedRuntimeException`
	`DatabaseConnectException`
	`DatabaseDisconnectException`
	`DatabaseNotConnectedException`
	`NotInTransactionException`
	`ObjectNameNotUniqueRuntimeException`
	`TransactionDatabaseClosedException`
	`TransactionDatabaseNotFoundException`

* JVI specific and not part of the ODMG 2.0 Java standard.

What's in this chapter?

This chapter contains explanations of basic concepts and shows you how to use ODMG JVI based on the concepts. The following table lists each concept and the associated tasks:

Basic Concepts	**Tasks**
Database class	Opening and closing a database
Database class	Using named objects
Transaction class	Associating a transaction with a database
	Instantiating a transaction
	Securing a transaction
	Starting a transaction
	Closing a transaction
	Using threads and transactions
	Accessing the Versant session
	Using multiple databases
	Using locks
Exception class	Understanding inconsistencies in the ODMG standard
Queries	Explanation only

For more information, on the concepts of persistent class categories, transitive persistence, and enhancement please refer to Chapter 3 “Transparent JVI” on page 103.

DATABASE CLASS

Overview

com.versant.odmg.Database

Class `Database` encapsulates a database accessed in the ODMG Java program.

The `Database` object is transient. You must open databases before you start any transactions that use the database and close the databases when you finish.

To explain the database class, this section describes:

- Opening and closing a database
- Using named objects

Opening and closing a database

Open a database

To access a database, you must first obtain an instance of the `Database`.

You can obtain an instance by invoking the static `open()` factory method on the `Database` class. This method takes the database name and the mode in which to open it as arguments.

For example:

```
Database db1;
try {
 // Open database called 'mydb' with read-write privileges
 db1 = Database.open("mydb", Database.openReadWrite);
} catch (ODMGException e) {
 if (e instance of DatabaseOpenException) {
  System.out.println("mydb is already open! " + e);
 }
 else if (e instance of DatabaseNotFoundException) {
  System.out.println("Database 'mydb' not found! " + e);
 }
 else {
  System.out.println("ODMGException in Database.open: " + e);
 }
}
```

Close a database

After you finish using a database, you must close it. Closing a database cleans up the connection to the database. You cannot close a database if any transactions are open. If any transactions are open, the following runtime exception occurs: `TransactionInProgressException`.

After you close a database, further attempts to access objects in the database will fail.

For example:

```
try {
 // db1 is an instance of Database
 db1.close();
} catch (ODMGException e) {
 if (e instance of DatabaseClosedException) {
  System.out.println("db1 is already closed! " + e);
 }
 else {
  System.out.println("ODMGException in db1.close: " + e);
 }
```

Use named objects

The `bind()`, `unbind()` and `lookup()` methods on the `Database` class allow you to manipulate objects by name.

To explain how you can use named objects, this section describes:

- How to bind an object to a name
- How to unbind an object from a name
- How to look up an object by name

Bind an object to a name

The `bind()` method can make a previously transient object persistent by binding it to a name in the database. You can bind one object to multiple names in the database. However, a single name can refer to only a unique persistent object. Binding an object to a name that already has an associated persistent object throws the following exception: ObjectNameNotUniqueRuntimeException.

For example

```
// db1 is an instance of Database
// p is an instance of a persistent capable class, Person
Person p1 = new Person("Jeff Stanzen", 10280);

try {
 db1.bind(p1, "HeadOfDepartment");
} catch (ObjectNameNotUniqueRuntimeException e) {
  System.out.println("Object with name " + "'HeadOfDepartment'
exists:" + e);
 }
```

An inconsistency appears in the ODMG 2.0 Java binding specifications. The signature of the Database.bind() method does not specify an ODMGException or any other exception that must be caught.

The ObjectNameNotUniqueException is a subclass of ODMGException. This exception occurs when you bind an object with an existing object association. JVI defines and throws an ObjectNameNotUniqueRuntimeException in this situation. This exception ensures that the method signature of bind() remains the same as the ODMG specification.

Unbind an object from a name

The unbind() method drops the name association of a persistent object. If an object has not been bound to the specified name, it throws the following exception: ObjectNameNotFoundException.

For example:

```
// db1 is an instance of Database
try {
 db1.unbind("HeadOfDepartment");
} catch (ODMGException e) {
 if (e instanceof ObjectNameNotFoundException) {
  System.out.println("Object with name " +
    "'HeadOfDepartment' not found");
 }
}
```

Look up an object by name

The `lookup()` method returns the object bound to a particular name. If no object is bound to the specified name, it generates the following exception: `ObjectNameNotFoundException`.

For example:

```
// db1 is an instance of Database
try {
  Person p = (Person) db1.lookup("HeadOfDepartment");
  } catch (ODMGException e) {
  if (e instanceof ObjectNameNotFoundException) {
  System.out.println("Object with name " +
  "'HeadOfDepartment' not found");
  }
}
```

If the database in the `bind()`, `unbind()` and `lookup()` methods has already been closed, then invoking these methods throws the following exception: `DatabaseClosedRuntimeException.` JVI defines this exception and it is not part of the ODMG Java binding specifications.

The `DatabaseClosedException` is a subclass of `ODMGException.` The methods `bind()`, `unbind()` and `lookup()` do not throw this exception.

Also, for the `bind()`, `unbind()` and `lookup()` methods to work, it is necessary to have an open transaction associated with the database. Otherwise, each of the methods throws the following runtime exception: `TransactionNotInProgressException.`

TRANSACTION CLASS

Overview

Transactions define the units of work in the ODMG model. You can start, commit, abort, and check point a transaction. You must open a transaction to access, create, and modify persistent objects or their fields.

In JVI, the ODMG `Transaction` class has a close relationship with a Versant Transparent Session (`com.versant.trans.TransSession`). Each `Transaction` instance corresponds to a unique Versant Transparent Session instance.

Associate a transaction with a database

Concepts

Before you can instantiate a `Transaction`, a database must be opened earlier in the executing thread, or somewhere in the Java program.

Creation of the underlying Versant session requires association with a database. A Versant session is created when a `Transaction` object is instantiated. This database associated with the session (and transaction) by default is the Session Default Database. When you create a persistent object in a transaction, the object is written to the Session Default Database when the transaction is committed.

If you have created your `Transaction` object with a constructor that takes a `Database` instance as an argument, then the Session Default Database for that transaction is the specified database. Otherwise, a default database selection policy determines the Session Default Database of a transaction.

JVI uses the terms Thread Default Database and Process Default Database to identify the default database associated with a transaction.

The first database you open in a thread is the Thread Default Database for that thread. It remains open until you close the database. After you close the database, the Thread Default Database is set to empty. The empty Thread Default Database is automatically set when you open another database in this thread.

Similarly, you also set the Process Default Database to the first database you open in the Java program. After you close the database, the Process Default Database is set to empty. The empty Process Default Database is automatically set to the next database you open in the Java program.

A `Transaction` constructor that does not explicitly take a `Database` instance as argument uses the following rules to determine which database is associated with the transaction:

1. The first database associated with the transaction is the Thread Default Database, if it exists for the executing thread.
2. If the Thread Default Database is unset, but the Process Default Database exists, then the Process Default Database is used as the associated database.
3. If both the Thread and Process Default Databases are unset, then a runtime `TransactionDatabaseNotFoundException` is thrown.

To explain how to associate a transaction with a database, this section describes:

- How to use a Thread Default Database
- How to use a Process Default Database with a transaction

Use a thread default database

You can use a Thread Default Database with a transaction.

For example:

```
// Thread T
Database d1;
try {
 d1 = Database.open("threaddb", Database.openReadWrite);
} catch (ODMGException e) {
 System.out.println("Exception : " + e + " in Database.open for
          'threaddb'");
 System.exit(1);
}
Transaction tx = new Transaction();
// tx is now associated with Thread Default Database 'threaddb'
```

Use a process default database with a transaction

You can use a Process Default Database with a transaction.

For example:

```
// Thread Ta
Database d1;
void initdb() {
 try {
   d1 = Database.open("processdb", Database.openReadWrite);
 } catch (ODMGException e) {
  System.out.println("Exception : " + e + " in Database.open for
           'processdb'");
  System.exit(1);
 }
}

// Thread Tb
// Executes after Thread Ta executes initDb ()

Transaction tx = new Transaction();
// tx is now associated with Process Default
// Database 'processdb'
```

Instantiate a Transaction

Instantiating a `Transaction` before you open a `Database` can result in the following runtime exception: `TransactionDatabaseNotFoundException`. In general, you must instantiate `Transactions` in your program only after opening a `Database`. The following example shows how an exception is thrown when a `Transaction` is instantiated before a `Database` is opened.

For example:

```
// Thread Tc
Transaction tx = new Transaction ();

// a TransactionDatabaseNotFoundException gets thrown

try {
  d1 = Database.open("processdb", Database.openReadWrite);
} catch (ODMGException e) {
 System.out.println("Exception : " + e + " in Database.open for
           'processdb'");
 System.exit(1);
}
```

NOTE:- A `Transaction` constructor that explicitly accepts a `Database` instance as an argument associates the specified database with that transaction. JVI encourages you to use this constructor. By doing so, you can avoid reliance on the default database model and make your program more deterministic and easier to understand and debug.

For example:

```
try {
  d1 = Database.open("mydb", Database.openReadWrite);
} catch (ODMGException e) {
 System.out.println("Exception : " + e + " in Database.open for
           'mydb');
 System.exit(1);

Transaction tx = new Transaction(d1);

//tx is now associated with 'mydb'
```

Secure a transaction

The `Transaction` class has constructors that accept a `Capability` instance as an argument. These constructors provide open "secure" transactions with the Versant database in accordance with the `Capability` based security model.

For more information refer to “Security” on page 151, in "Chapter 3 - Transparent JVI".

As discussed previously in "Associating a database", the use of secure transactions is orthogonal to the default database model.

Start and stop a transaction

Start a transaction

The `begin()` method on a `Transaction` instance starts (or opens) a transaction. The transaction is then open until you invoke either the `commit()` or `abort()` operations on the `Transaction` instance.

Before performing any database operations, a thread must explicitly create a `Transaction` object or associate itself with an existent `Transaction` instance by invoking the `join()` method on the `Transaction`. The `Transaction` must then be started.

Invoking `begin()` on a `Transaction` implicitly joins the thread to that transaction.

Close a transaction

A `Transaction` object closes after you invoke either a `commit()` or `abort()` method. Invoking the `commit()` method does the following:

- Commits all modifications to the database.
- Deletes objects marked for deletion.
- Releases all held locks.
- Closes the transaction.

The `Transaction.commit()` method commits the underlying Versant session.

Invoking the `abort()` method does the following:

- Aborts the current transaction.
- Abandons all persistent object modifications.
- Releases any associated database locks.

`Transaction.abort()` rolls back the underlying Versant session.

Use threads and transactions

Concepts

Before you use a `Transaction` instance, a thread must join the `Transaction` by invoking the `join()` method on the `Transaction`. A thread that opens a `Transaction` with the `begin()` method implicitly joins the `Transaction`. Before performing any work in a transaction, a thread needs to have joined the transaction. Otherwise, database kernel runtime exceptions can be thrown based on the operation being performed.

The following shows you how you can join a thread to a transaction:

Join a thread to a transaction

You can join a thread to a `Transaction`.

For example:

```
// Thread Td

Database d1;
Transaction tx;

void initTx() {
 try {
  d1 = Database.open("processdb", Database.openReadWrite);
 } catch (ODMGException e) {
  System.out.println("Exception : " + e + " in Database.open for
           'processdb');
  System.exit(1);
 }

 tx = new Transaction();
 tx.begin();
}

// Thread Te
// Executes after Thread Td executes init ()
tx.join();
```

```
// Thread Te can now work in Transaction tx.
```

A thread can be associated with only one transaction at a time. Thus, when you join a transaction, a previously joined transaction is implicitly detached. The detached transaction remains in its previous state, and detaching the thread does not alter the transaction state.

The `Transaction` class also exposes the `leave()` method. You can call this method on an instance to explicitly cause the thread to leave the `Transaction`.

Invoking `leave()` on a `Transaction` object not joined earlier throws the following runtime exception: `NotInTransactionException`.

The `Transaction` class also contains a `staticCurrent()` method. This method returns the `Transaction` object associated with the calling thread.

The ODMG Java binding specifications outline three typical ways users can use multiple threads with transactions:

- An application program could have a single `Transaction` instance used in a single thread.

  This programming model is very simple. The program requires no synchronization for database and transaction operations and no requirement for threads to join and leave transactions.
- An application program could have multiple threads. In addition, each multiple thread in the program can have a separate `Transaction` instance.

  In this programming model, the threads do not share the `Transaction` instances. Therefore, a clean separation of transaction-related data between the threads occurs. You can use this model to write applications that implement a service accessed by multiple clients on a network.

NOTE:- When you use this model, be careful not to pass any database-related objects used in one thread to another thread in a different transaction.

- Multiple threads could share one or more transactions.

  This programming model is the most complex scenario. If a `Transaction` instance is associated at the same time with multiple threads, then all these threads are affected by data or transaction operations. You must explicitly synchronize the threads that share the `Transaction` instances with appropriate Java synchronization constructs. Synchronization allows you to provide concurrency control in your application.

Access the Versant Session

You can access the underlying Versant Transparent Session that lies beneath a Transaction by invoking the session() method on the Transaction instance. This method returns an ODMGSession instance. Recall that each Transaction instance corresponds to a unique Versant Transparent Session instance.

ODMGSession is a subclass of com.versant.trans.TransSession. Hence, all the public methods on TransSession can be invoked on the ODMGSession instance. You can invoke methods on ODMGSession based on whether the associated Transaction is open. Invoking methods on ODMGSession can also influence the Transaction open state.

For example, if the associated Transaction instance has not begun and you invoke the commitAndCleanCod() method on ODMGSession, the following ODMGRuntimeException occurs: TransactionNotInProgressException. The associated Transaction is closed after the method executes.

NOTE:- The commitAndCleanCod() method on ODMGSession is an advanced Versant specific memory management API used with a database commit.

The following ODMGSession methods close the associated Transaction instance:

```
commit()
commitAndCleanCod()
commitAndRetain()
commitAndRetain(Object[])
commitAndRetainSchema()
rollback()
rollbackAndRetain()
rollbackAndRetain(Object[])
```

Behavior of commit_delete

If the server profile parameter commit_delete in turned OFF, the delete APIs deleteObject() and groupDeleteObjects() write directly to the server. The objects are deleted immediately.

If commit_delete is set to ON, the delete APIs will mark the objects for deletion in the database server. The objects are immediately released from the front end cache. The server maintains a list of objects marked for deletion, and these are deleted when a commit is performed. In case of a rollback the objects are unmarked and they retain their original status.

In addition, the ODMGSession class provides methods for transactional support to multiple databases.

Use multiple databases

The JVI `Transaction` class further extends the specifications provided in the ODMG 2.0 Java bindings by supporting transactional access to multiple databases.

`ODMGSession` contains the following two methods to attach and detach a specified `Database` object to and from a `Transaction` instance respectively:

```
connectDatabase(Database)
disconnectDatabase(Database)
```

To connect to a database, the `Database` instance must have been previously opened. Otherwise, the following exception occurs: `DatabaseClosedException`. In addition, the database should not have already been connected to the same `Transaction` earlier. Otherwise, the following runtime exception occurs: `DatabaseConnectException`.

To disconnect from a database, the `Database` instance must have also been previously opened. Otherwise, the following exception occurs: a `DatabaseClosedException`.In addition, the database should not have already been disconnected from the `Transaction` or not connected to the `Transaction` at all. Otherwise, the following runtime exception occurs: `DatabaseNotConnectedException`.

The database disconnected from the `Transaction` cannot be the initial database associated with the `Transaction` because this forms the session database of the underlying Versant Session. You must associate `Transactions` (and their underlying `Sessions`) with a database.

Use locks

The `Transaction` class contains the `lock()` method. This method takes the following two arguments:

- An object on which the lock is sought.
- A lock mode used in upgrading the lock on an object.

  The lock mode can be either `READ`, `WRITE` or `UPGRADE`.

Invoking the `lock()` method upgrades the lock on the given object to the specified level, only if it is not already at or above that level. If the lock cannot be granted, the method throws the following exception: `LockNotGrantedException.` This method also provides a Versant error message that explains why the lock could not be granted .

Note that a read lock is implicitly obtained on an object the first time the fields are accessed in a transaction. Similarly, a write lock is obtained when a field is modified.

An update lock allows you to read an object and get the next available write lock on it. An update lock is useful if you want to look at an object right away and update it later. Possessing an update lock on an object guarantees the blocking of all other requests for an update or write lock, but permits other requests for read locks on the object.

COLLECTION CLASSES

Overview

ODMG JVI supports two different collection standards based on the ODMG Java Binding:

- ODMG 2.0 – The original ODMG Java Binding, this release offers ODMG-specific `Collection`, `Set`, `Bag` and `List` interfaces. The associated classes and interfaces for JVI are in the `com.versant.odmg` package.
- ODMG 2.1 – This newer version of the ODMG Java Binding specifies collection classes and interfaces based on the JDK 1.2 collections package (`java.util`). The associated classes and interfaces for JVI are in the `com.versant.odmg3` package.

ODMG 2.0 Collection Interfaces

The ODMG 2.0 Java Binding Specification mandates that a vendor package must provide the following collection interfaces: `Collection, Set, Bag` and `List`.

JVI provides all four interfaces in the `com.versant.odmg` package.

ODMG 2.1 Collections

The ODMG 2.1 Java Binding specification adds support for JDK 1.2 collection interfaces. These interfaces are `DArray, DBag, DCollection, DList, DSet and DMap`. Note that these ODMG-specific interfaces extend the base JDK 1.2 interfaces `Collection, List`, `Set and Map`. In addition, an ODMG 2.1 compliant interface needs to support the concrete classes `ArrayOfObject, SetOfObject, BagOfObject, ListOfObject and MapOfObject.`

To support both the ODMG 2.0 and ODMG 2.1 specifications, JVI places the ODMG 2.1 classes and interfaces in the `com.versant.odmg3` package. The package is named “`odmg3`” in anticipation of the ODMG 3.0 release, which should support the same set of collection classes.

The elements that can be inserted in these collection classes could be of any type. These elements could either be persistent objects with their own database LOID, or objects that do not belong to a persistence category. If the elements are not instances of classes designated as Persistent Capable or Persistent Always, their classes must implement the `java.io.Seri-`

`alizable` or `java.io.Externalizable` interface. These objects are serialized when written to the database.

EXCEPTION CLASSES

Overview

ODMG-defined exceptions have two categories:

- `ODMGException` and its subclasses define exceptions that explicitly need to be caught in application code, or declared as thrown in method signatures.
- `ODMGRuntimeException` and its subclasses define runtime exceptions that do not necessarily have to be caught in application code.

Database kernel errors are thrown as `VExceptions`. For example:

```
Versant error number 5006, OB_NO_SUCH_OBJECT: Cannot find the object
```

`VException` is part of the `com.versant.fund` package, but it is a subclass of `ODMGRuntimeException`. Hence, all the `VExceptions` thrown (which include all database kernel related exception conditions) can be caught as `ODMGRuntimeExceptions`.

Inconsistencies in the ODMG Standard

The ODMG 2.0 Java binding standard does not specify the `bind(), lookup()` and `unbind()` methods of `Database` as throwing any non-runtime exceptions. However, the `DatabaseClosedException` is thrown whenever a database is accessed after it has been closed. This exception is a non-runtime exception.

To avoid altering method signatures from the ODMG specifications, JVI introduces a `DatabaseClosedRuntimeException`, a subclass of `ODMGRuntimeException`, which is the runtime equivalent of `DatabaseClosedException`.

In addition, the `ObjectNameNotUniqueException` should be thrown when a name being bound to is not unique. This exception is not a runtime exception, yet the method signature of `Database bind` does not include the throwing of this exception.

To avoid altering the `bind()` method signature, JVI introduces an `ObjectNameNotUniqueRuntimeException`, a subclass of `ODMGRuntimeException.` This exception is the runtime equivalent of `ObjectNameNotUniqueException`.

QUERIES

JVI does not support the Object Query Language, OQL, specified in the ODMG specifications. Instead, JVI provides APIs to the Versant Query Language capability, VQL, in the transparent binding.

For more information, on how to use VQL in your ODMG applications, please refer to the Chapter 3 “Transparent JVI” on page 103.

CHAPTER 5

Utility JVI Reference

This Chapter gives detailed explanation about the utility classes in “Utility JVI”.

The Chapter explains the following in detail:

- Introduction to utilities
- Collections
- Statistics

INTRODUCTION

Utility JVI contains utility classes to access database environment and administrative information, and frequently used collection classes. You can use Utility JVI from both Transparent and ODMG JVI.

What's in this chapter?

This chapter contains explanations of specific concepts and shows you how to use Utility JVI based on the concepts. The following lists each concept and the associated tasks:

Concepts	Tasks
Database Utilities	**Using database utilities** • How to create a database • How to add volume • How to get database information • How to get user information • How to add and remove users • How to set the database mode • How to get database environment information • How to get database mode • How to compare two databases • How to stop database • How to remove a database • How to copy a database • How to convert a database • How to create an osc-dbid file or make/delete entries in the osc-dbid file • How to backup / restore a database • How to start/stop/disable a RollforwardArchive process. • How to get information about a backup file • How to get a list of backups • How to get vbackup progress status.

Collections

See Chapter 3 “Transparent JVI” on page 103 and Chapter 4 “ODMG JVI” on page 247.

Statistics

Using interface methods to collect statistics

- How to turn on statistics
- How to turn off statistics
- How to get a current values list of active statistics
- How to get information about locks, transactions and connections
- How to use automatic statistics collection

To Create a Database

To create a database, you first need to make a database directory and then create the database.

makeDB()

You can use the `makeDB()` method to make a database directory.

For example, to make a group database and its owner called Mike, in the `VERSANT_DB` directory,

```
Properties p = new Properties();
p.put ("-g","");
p.put ("-owner","Mike");
DBUtility.makeDB("mikedb", p);
```

NOTE:- If the user wants to set a DBA password for the new database, he will have to add “–password” as a key and the DBA password as the value for this key, in the properties parameter.

If a database is associated with a DBA password, then all DBA utility APIs on this database will require the password to be specified.

createDB()

You can use the createDB() method to create a database.

For example, to create a database,

```
DBUtility.createDB("mikedb");
```

If a database is associated with a DBA password, then use following API to create a database:

```
DBUtility.createDB("mikedb", "Password");
```

To Add a volume

addVol()

You can use the addVol() method to add a volume to the database. Adding a volume increases database storage capacity.

For example, to add an extra volume file "extraVolume" to path c:\versant\db\javadb with size 10M and extent size of 2 pages:

```
Properties p = new Properties();
p.put ("-v","extraVolume");
p.put ("-p","c:\versant\db\javadb");
p.put ("-s","10M");
p.put ("-e", "2");
DBUtility.addVol(dbname, p);
```

NOTE:- If the database is associated with a DBA password, then you will require to add “–password” as a key and the DBA password as the value for this key, in the properties parameter.

To Get Database Information

To explain how to get database information, this section describes the following methods on class DBListInfo.

```
dbList()
dbListOwnedBy()
namedDBInfo()
```

```
dbListInVersantDBDirectory()
```

DBListInfo Class

The `DBListInfo` class contains attributes that describe a database and the methods that return their values. For example, you can obtain the:

- Database name with the `getDBName()` method.
- Database ID with the `getDBID()` method.
- Database type with the `getDBType()` method.
- Database owner with the `getDBOwner()` method.
- Time the database was created with the `getDBCreateTime()` method.
- Database version with the `getDBVersion()` method.

dbList()

You can use the `dbList()` method to obtain database information on all databases or on all databases of a given host.

For example, to return a list of database information from the host named `myMachine`:

```
DBListInfo[] myInfo = DBUtility.dbList("myMachine");
```

To return a list of database information on all databases on the executing machine:

```
DBListInfo[] myInfo = DBUtility.dbList();
```

dbListOwnedBy()

You can use the `dbListOwnedBy()` method to return a list of database information for all the databases owned by a given user.

For example, to return a list of database objects owned by user `Mike`:

```
DBListInfo[] myInfo = DBUtility.dbListOwnedBy("Mike");
```

To return a list of database objects from the host named `myMachine` and owned by user `Mike`:

```
DBListInfo[] myInfo = DBUtility.dbListOwnedBy("myMachine", "Mike");
```

namedDBInfo()

You can use the `namedDBInfo()` method to return database information on a given database.

For example, to return a list of database information for the database named `javadb`:

```
DBListInfo[] myInfo = DBUtility.namedDBInfo("javadb");
```

dbListInVersantDBDirectory()

You can use the `dbListInVersantDBDirectory()` method to return database information on all the existing databases (including empty database directories) in the `VERSANT_DB` directory.

For example:

```
DBListInfo[] myInfo = DBUtility.dbListInVersantDBDirectory();
```

To Get User Information

getDBUsers()

You can use the `getDBUsers()` method to get information about users who have access to a database.

For example, to return a list of users with access to the database `javadb`:

```
DBListInfo[] myInfo = DBUtility.getDBUsers("javadb");
```

To Add and Remove Users

addDBUser() / removeDBUser()

You can use the `addDBUser()` and `removeDBUser()` methods to add and remove users from a database access list.

For example, to add user `Mike` to the database `javadb` access list:

```
DBUtility.addDBUser("javadb", "Mike", "password", "accessMode")
```

To remove user `Mike` from the database `javadb` access list:

```
DBUtility.removeDBUser("javadb", "Mike");
```

NOTE:- If a database is associated with a DBA password, then use following API to add and remove a user from the database:

```
DBUtility.addDBUser("javadb", "Mike", "password", "accessMode", "dba
 password");
DBUtility.removeDBUser("javadb", "Mike","dba password");
```

To Set the Database Mode

dbInfo()

You can use the `dbInfo()` method to set the database mode. The different modes are multi-user, unstartable, DBA-only single-connection and DBA only multi-connection.

For example, to set the database `javadb` to multi-user mode:

```
DBUtility.dbInfo("javadb", "-m");
```

NOTE:- If a database is associated with a DBA password, then the following option should be used to set database to multiuser mode:

```
DBUtility.dbInfo("javadb", " –m –password dbapassword" );
```

To Get Database Environment Information

dbEnvInfo()

You can use `dbEnvInfo()` to get information about the database environment.

For example, to get an instance of `DBEnvInfo`, similar to using the Versant utility command `oscp -i`:

```
DBUtility.dbEnvInfo()
```

To Get the Database Mode

nameDBMode()

You can use the `nameDBMode()` method to get the database.

For example, to get the mode of the database `javadb` :

```
DBUtility.getNameDbMode("javadb);
```

To Compare the Databases

comparedb()

You can use `compareDb()` method to compare the databases `dbName1` and `dbName2` and return as array of objects of `CompareDBResult`.

For example, to compare two databases,

```
Properties p = newProperties();
CompareDBResult[] result = DBUtility.compareDb (dbName1, dbName2, p);
```

To do value based comparison,

```
Properties p = new Properties();
p.put ("-value," " ");
CompareDBResult [] = DBUtility.compareDb (“dbName1”, “dbName2”, p);
```

NOTE:- If the database is associated with a DBA password, then you will require to add “–password” as a key and the DBA password as the value for this key, in the properties parameter.

compareDBs()

You can use `compareDBs()` method to compare the databases `jdb` and `jdb2` which supports newly added options.

```
Properties p = new Properties();
```

```
p.put ("-value", " ");
p.put ("-fullcompare", " ");
DBCompareResult  result   = DBUtility.compareDBs ("jdb","jdb2" , p);
if (result.getStatus() ==
 com.versant.fund.Constants.DATABASES_ARE_IDENTICAL)
{
  System.out.println("Database are identical");
else {
  System.out.println(" Databases are not identical");
  Vector loid1 = result.getLoidsDB1();
  Vector loid2 = result.getLoidsDB2();
  Vector loidsVal = result.getLoidsByValue(); // only if "-value" is
  supplied
}
```

NOTE:- If the database is associated with a DBA password, then you will require to add "–password" as a key and the DBA password as the value for this key, in the properties parameter.

compareDBsAdvanced()

You can use `compareDBsAdvanced()` method to compare the databases `jdb` and `jdb2` which supports `–advanced` option.

For example

```
Properties p = new Properties();
p.put ("-value", " ");
String classes = "com.vin.Person" + " " + "com.vin.Employee";
p.put ("-classes", classes);
int status = DBUtility.compareDBsAdvanced("jdb", "jdb2" , p, "/tmp/
loids");
if (status == com.versant.fund.Constants.DATABASES_ARE_IDENTICAL)
  System.out.println(" Databases are identical ");
else {
  System.out.println(" Databases are not identical");
  /* /tmp/loids will contain the differed loids */
}
```

NOTE:- If the database is associated with a DBA password, then you will require to add "–password" as a key and the DBA password as the value for this key, in the properties parameter.

To Stop the Database

stopDB()

You can use the `stopDB()` method to stop a database and remove all corresponding database resources in memory.

For example, to stop database `javadb`

```
Properties properties = new Properties();
properties.put("-f", " ");
DBUtility.stopDB(“dbName”, properties);
```

NOTE:- If the database is associated with a DBA password, then you will require to add “–password” as a key and the DBA password as the value for this key, in the properties parameter.

To Remove the Database

removeDB()

You can use the `removeDB()` method to stop the database if it is running, destroy and remove all volumes of the database `dbname`, and delete the database from the system database identifier file `osc-dbid`.

For example

```
Properties properties = new Properties();
properties.put("-f", " ");
DBUtility.removeDB(“dbName”, properties);
```

NOTE:- If the database is associated with a DBA password, then you will require to add “–password” as a key and the DBA password as the value for this key, in the properties parameter.

To Convert the Database

convertDB()

You can use the method `convertDB()` to convert an existing database created with a prior Versant Release to work with the current Versant Release and return an error if not successful.

For example

```
Properties properties = new Properties();
DBUtility.convertDB(dbName, properties);
```

The convertDB API is always force executed with the option -noprint.

This enforces the silent mode.

NOTE:- Use of `convertDB()` may lead to deletion of some indexes.

If some indexes have been deleted as a result of executing `convertDB`, the user will find the deleted index info in "`deletedindex.dat`" in the database directory.
A shell script "`recreateIndex.sh`" (or batch file "`recreateIndex.bat`" on Windows) is also generated, in the database directory, that can be executed later to create the deleted indexes.

For more information, please refer to Section on Convertdb utility in the Chapter "Database Utilities" in the *Versant Database Administration Manual*.

To Copy a Database

copyDB()

You can use the `copydb()` method to copy objects from one database to another database.

For example

```
Properties properties = new Properties();
properties.put("-nolock", " ");
DBUtility.copyDB("myDB", "myDB1", properties);
```

NOTE:- If the database is associated with a DBA password, then you will require to add "–password" as a key and the DBA password as the value for this key, in the properties parameter.

To Create an osc-dbid File

dbid()

You can use the `dbid()` method to create an `osc-dbid` file in Versant DB path and create/ delete entry for the database from the `osc-dbid` file.

For example

```
Properties properties = new Properties();
properties.put("-C", "50");
properties.put("-t", "1");
DBUtility.dbid("myDB", properties);
```

For Synchronous Database Replication

ftstoolEnablePolling()

When Synchronous database replication (FTS) is used, this API can be used to turn automatic synchronization on.

For example :

```
DBUtility.ftstoolEnablePolling("mydb");
```

NOTE:- If a database is associated with a DBA password, then following API should be used to turn automatic synchronization on.

For example :

```
DBUtility.ftstoolEnablePolling("mydb", "DBAPassword");
```

ftstoolDisablePolling()

When Synchronous database replication (FTS) is used, this API can be used to turn automatic synchronization off.

For example :

```
DBUtility.ftstoolDisablePolling("mydb");
```

NOTE:- If a database is associated with a DBA password, then following API should be used to turn automatic synchronization off.

For example :

```
DBUtility.ftstoolDisablePolling("mydb", "DBAPassword");
```

ftstoolStopSync()

When Synchronous database replication (FTS) is used, this API can be used to stop the synchronization of the replicated database.

For example :

```
DBUtility.ftstoolStopSync("mydb");
```

NOTE:- If a database is associated with a DBA password, then following API should be used to stop the synchronization of the replicated database.

For example :

```
DBUtility.ftstoolStopSync("mydb", "DBAPassword");
```

ftstoolDisableReplication()

When Synchronous database replication (FTS) is used, this API can be used to turn automatic synchronization off.

For example

```
DBUtility.ftstoolDisableReplication("myDB");
```

NOTE:- If a database is associated with a DBA password, then the following API should be used to turn automatic synchronization off.

```
DBUtility.ftstoolDisableReplication("myDB", "dbapassword");
```

ftstoolForceDisableReplication()

When Synchronous database replication (FTS) is used, this API can be used to break the replication of the databases in a FTS pair.

For example

```
DBUtility.ftstoolForceDisableReplication("myDB");
```

NOTE:- If a database is associated with a DBA password, then the following API should be used to break the replication of the databases in a FTS pair.

```
DBUtility.ftstoolForceDisableReplication("myDB", "dbapassword");
```

ftstoolEnable()

When Synchronous database replication (FTS) is used, this API can be used to restart the down database and to re-sync after fail-over

For example

```
DBUtility.ftstoolEnable("myDB");
```

NOTE:- If a database is associated with a DBA password, then the following API should be used to restart the down database and to re-sync after fail-over.

```
DBUtility.ftstoolEnable("myDB", "dbapassword");
```

ftstoolGetReplicationStatus()

When Synchronous database replication (FTS) is used, you can use `ftstoolGetReplicationStatus()` to get the state of the database and polling process and return as object of class `FTSStatus`.

For example.

```
FTSStaus status = DBUtility.ftstoolGetReplicationStatus("myDB");
```

NOTE:- If a database is associated with a DBA password, then the following API should be used to get the state of the database and polling process.

```
FTSStatus status = DBUtility. ftstoolGetReplicationSta
tus("myDB","dbapassword");
```

Database Backups

For Backing up a Database

Vbackup utility is used for backing up a database.

To manage the database backup, restore and roll forward archiving operations through JVI, a new class `VBackupManager` has been introduced to the utility classes.

`BackupListner`, `RestoreListner` and `StartRollforwardArchivinglistner` are used to track the progress of the operations.

`BackupListner` is to signal and recover from different recoverable error conditions while backing up the database.

`RestoreListner` is to signal and recover from different recoverable error conditions and warnings while restoring database.

`StartRollforwardArchivinglistner` is to signal and recover from different recoverable error conditions and warnings while logging database.

The following methods of class `VbackupManager` can be used for the Backup/Restore of a database.

backup()

You can use the `backup()` method to take the backup of a database.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
Vbm.backup(backupListner, options);
```

For more information on `BackupListner` class and `Options` that can be passed for backup operations, please refer to Javadocs.

For Restoring a Database

restore()

You can use the restore method to restore the database.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
Vbm.restore(restorelistner, options);
```

For more information on `RestoreListner` class and `Options` that can be passed for restore operations, please refer to Javadocs.

For RollForwardArchiving on a Database

You can Start, Stop and Disable the rollforwardarchiving.

startRollforwardArchiving ()

You can use this method to start rollforwardarchiving while backing up the database.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
Vbm.startRollforwardArchiving (startRollforwardArchivinglistner,
 options);
```

For more information on `RollforwardArchiving` class and the `Options` that can be passed for Rollforward operations, please refer to Javadocs.

stopRollforwardArchiving()

You can use this method to stop rollforwardarchiving while backing up the database.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
Vbm.stopRollforwardArchiving ();
```

DisableRollforwardArchiving()

You can use this method to disable rollforward archiving while backing up the database.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
Vbm.DisableRollforwardArchiving ();
```

For Getting Information about a Backup File

GetBackupInformation()

You can use this method to provide information about the contents of backup file/s of the given database.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
Iterator iterbackupinformation = Vbm.getBackupInformation ();
```

`BackupInformation` class stores the information of the backups taken on the databases.

For more information, please refer to the Javadocs.

For Getting a List of Backups

GetListOfBackups()

You can use this method to list the set of backups necessary to restore a database from the given `deviceName`.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
DeviceInformation devinfo = Vbm.GetListOfBackup (deviceName);
```

`DeviceInformation` class stores the information of the device on which backup has been taken.

For more information, on `DeviceInformation` class and its `options`, please refer to Javadocs.

For Getting vbackup Progress Status

GetVBackupProgressStatus()

You can use this method to get the status of the backup /restore process.

For example

```
VbackupManager vbm = new VbackupManager (dbname);
VbackupProgressStatus vbps = Vbm.getVabckupProgressStatus ();
```

`VBackupProgressStatus` stores the backup /restore progress status such as mode of the `vbackup`, level of the `vbackup` process etc.

For more information on `VBackupProgressStatus` class and its `options`, please refer to Javadocs.

SUPPORT FOR JAVA COLLECTIONS

JVI supports the following persistent versions of the standard Java collections viz. Vector and Hashtable in the `com.versant.util` package:

```
VVector
DVector
VHashtable
LargeVector
```

For more information refer to “Collections” on page 150, in "Chapter 3 - Transparent JVI".

JVI supports ODMG Collections in ODMG package `com.versant.odmg3` which are as follows:

```
ListOfObject
SetOfObject
BagOfObject
MapOfObject
```

For more information refer to “Collection Classes” on page 264, in "Chapter 4 - ODMG JVI".

DATABASE STATISTICS

Statistics provides profiling information about the database operations. This information helps measure, analyze and tune application performance.

This section describes the following:

- Statistics types
- Ways to access statistics

Statistics Types

You can use the following types of statistics to collect information:

Session statistics

You can monitor session information in a smilar way as you collect application process information.

When monitoring application activity you can get information such as the number of objects read in the object cache.

Database connection statistics

You can monitor connections made to a database, such as locks or time spent on a certain database operation.

Latch statistics

You can monitor latches per database or per connection made to a database.

Derived statistics

You can derive your own statistics with combinations of other statistics, numerical operators and statistical functions.

Ways to Access Statistics

You can access statistics in the following ways:

1. Direct connection using `vstats` utility
2. Interface methods to obtain statistics
3. Automatic profiling

For more information, on each statistic functionality, please refer to Chapter "Statistics Collection" in the *Versant Database Fundamental Manual*.

Using vstats utility

For more information on vstats utility, please refer to Chapter "Database utilities" in the *Versant Database Administration Manual*.

Using Interface methods to Collect Statistics

To use interface methods to collect statistics, this section describes the following:

- How to turn on statistics
- How to turn off statistics
- How to get the current values list of active statistics
- How to get information about locks, transactions, and connections

Turn on statistics

Assuming the application is in an active session on a database named "db", the following code segment turns on statistics for the:

- current connection,
- number of objects read from database server, and
- number of objects written to the database server.

```
int[] stats = new int[2];
stats[0] = VStatsNames.STAT_FE_WRITES;
stats[1] = VStatsNames.STAT_FE_READS;
String dbName = "db";
```

```
int conn_id = VStatsNames.STAT_THIS_CONNECTION;
VStatistics.turnOnCollectionOfStatistics(stats, dbName, conn_id);
```

Turn off statistics

The following turns off statistics collection:

```
VStatistics.turnOffCollectionOfStatistics(stats, dbName, conn_id);
```

Getting a current values list of active statistics

You can get a list of current values of active statistics.

For example:

```
double[] stats_result = VStatistics.getStatistics(
stats,
dbName,
conn_id);
```

Get information about locks, transactions, and connections

You can use the `getActivity()` method on the `VStatistics` class to get information about locks, transactions and connections associated with a particular database.

The following code segment gets information about all locked objects, all transactions and all connections of the database named "`db`".

For example:

```
Object[][] result = VStatistics.getActivityInfo(
   "db",
   null,
   null,
   null);
DBLockInfo[] lockInfo_result;
lockInfo_result = (DBLockInfo[])(result[0]);
DBTransactionInfo[] transInfo_result;
transInfo_result = (DBTransactionInfo[])(result[1]);
DBConnectInfo[] connectInfo_result;
connectInfo_result = (DBConnectInfo[])(result[2]);
```

Instead of passing `null` as parameter value to limit the scope of the returned information, you can specify an array of `DBLockInfo`, `DBTransactionInfo` or `DBConnectInfo`.

In the following code segment, only information about the first connection is returned:

```
DBConnectInfo[] connectInfo = new DBConnectInfo[1];
connectInfo[0] = connectInfo_result[0];
Object[][] result = VStatistics.getActivityInfo(
   "db",
   null,
   null,
   connectInfo);
```

Use Automatic Statistics Collection

You can turn on automatic collection of statistics.

For example:

```
VStatistics.beginAutoCollectionOfAllStatistics(
   "db",
   0,
   null,
   "output");
```

You can turn off automatic collection of statistics.

For example:

```
VStatistics.endAutoCollectionOfStatistics();
```

You can add collection points for your own methods.

For example:

```
public void dummy() {
   VStatistics.autoStatsEnteringFunction("Main.dummy()", null);
   // do something meaningful
   VStatistics.autoStatsExitingFunction("Main.dummy()", null);
}
```

CHAPTER 6

Event Notification

This Chapter gives detailed about the concepts of “Event Notification”.

The Chapter covers the following in detail:

- Overview
- Example
- Building JVI Event Clients

OVERVIEW

JVI Event Notification is a framework for distributing notifications of database-related events to one or more event clients. This chapter defines the basic concepts of event notification, describes the architecture of the system and explains a simple example application that uses event notification.

Basic Concepts

Event notification is designed around the concept of an event. An event represents a change in the database – such as an object being created, modified or deleted – and captures the relevant information associated with that change. This information includes the nature of the change, the object that was involved in the change and the transaction in which the change occurred. Your application can use this information in a variety of ways. It could, for example, notify the user of an important change in the database, refresh out-of-date objects that have been modified in another transaction, or make further changes in the database.

You register your application’s interest in database events by defining an event channel. An event channel is a simple abstraction for broadcasting event notifications. After an event channel has been created, any number of applications can “tune in” to this channel to receive the associated event notifications. Channels share a global namespace, so that your application can access a previously defined channel simply by knowing its name.

JVI Event Notification uses the JavaBeans event model to invoke registered objects when an event has occurred. This is the same model used by the Java windowing toolkit (AWT). This means that receiving notifications of important database events is no more difficult than checking for a mouse click! Any event listener objects that are added to a channel will have their event callbacks invoked for each event that is delivered on that channel.

Architecture of Event Notification

The JVI Event Notification architecture is composed of three basic components: the Versant database server, the event daemon and any number of event clients. The database server, in addition to its normal processing of database activity, maintains an event queue. The server accepts event registrations, which specify the kinds of database operations that should cause events to be generated. Each time a database operation that matches an event registration occurs, the server adds a new event to the event queue.

For more information, on event processing in the database server, please refer to Chapter "Event Notification" in the *Versant Database Fundamentals Manual*.

The database server is not responsible for delivering event notifications to interested clients. This task is delegated to a separate process, the event daemon. The event daemon is actually a database client, whose main responsibility is to poll the event queue and deliver the event notifications to event clients. This separation between database server and event daemon allows for more flexible event-based architectures; the database server provides the raw event functionality, allowing the event daemon to be customized for the application's needs. While some Versant users might choose to build their own, customized event daemons and clients, JVI Event Notification provides a simple, easy-to-use, packaged solution.

The event daemon used in JVI Event Notification handles the details of polling the event queue, and encapsulates Versant event registrations in a higher-level channel abstraction. These named channels allow multiple, disparate clients to share event registrations.

JVI Event Notification provides a customized delivery engine that implements an event transport protocol based on TCP/IP sockets. For each JVI event client, a separate socket connection is opened, and event notifications are shipped across this connection. An event is delivered to a client only if that client has registered interest for that event – that is, only if the client has subscribed to that channel.

The JVI event daemon can support multiple, distributed event clients. The event client listens for event notifications and distributes these events to any event listeners that have registered for events on the associated channel. The event client also provides channel management functions, allowing channels to be created, deleted, disabled and enabled. Although an event client is typically also a database client, this is not strictly necessary. The event daemon operates on behalf of the event client, handling all of the event-related communication with the database server.

Note that there is a one-to-one relationship between a Versant database server and an event daemon. Each event daemon can interact with only one server. Similarly, there is a one-to-many relationship between event daemon and event clients.

Although a single event daemon can service multiple event clients, each event client can work with only one daemon (and hence one database). However, if an application needs to receive events from multiple databases, it can simply create multiple event clients (even within a single Java virtual machine).

EXAMPLE

This section demonstrates how to use JVI Event Notification by describing a simple example that simulates a stock-trading application. It is not intended to be an accurate simulation of how stock exchange systems work, but it does show how event notification and persistent Java objects can work together. The source and related files for this example can be found in your JVI installation in the `demo/event/` directory.

The stock example consists of two separate applications, the `StockTrader` and the `StockWatcher`. Both of these applications use a small persistent class, the `StockQuote`. The `StockQuote` class contains two attributes: the name of the stock, and the current trading price.

```
class StockQuote
{
      String symbolName;
      double price;
}
```

The `StockTrader` is not an event-based application; it is a regular JVI client with a graphical user interface. It allows shares of stock to be bought or sold. When shares are bought, the price goes up. When shares are sold, the price goes down. (Ideally, you will buy low and sell high!)

The `StockWatcher` is both an event client and a JVI client. It displays the current prices for each of four different stocks in a window. Whenever there is a change (made by the `StockTrader`) to a `StockQuote` object, it refreshes its display to reflect the change. The `StockWatcher` receives the notification of these changes on a channel named “stock”. This channel will notify any listeners on this channel when there is a change in an instance of the `StockQuote` class. The “stock” channel is created by the first `StockWatcher` application that is executed; all subsequent executions of the `StockWatcher` will share this same channel.

Any number of `StockTrader` and `StockWatcher` applications can be running simultaneously. When any trade is placed by a `StockTrader`, all of the `StockWatcher` windows will automatically reflect the change in price. These applications can be running on the same machine or distributed across the network.

Running the Application

Before studying the implementation of the stock example, you should run it. This will allow you to relate the structure of the program to its behavior. The major steps in running the application are:

1. Prepare the database.
2. Start up the event daemon.
3. Compile and enhance the Java source files.
4. Run the `StockWatcher` and `StockTrader` applications.

To get started, you should make sure that you are operating in the directory containing the stock event example. In the following commands, replace “JVI” with the JVI installation directory:

```
cd JVI/demo/event/
```

NOTE:- If using Windows, replace the “/” characters with “\” in this path and in those shown below.

Step 1: Prepare the Database

For this example we will assume a database named “`eventdb`”. This database will contain the `StockQuote` objects, as well as the objects that maintain the persistent channel information. The database preparation consists of three steps:

1. Make the database directory.
2. Create the database files.
3. Define the channel schema in the database.

The commands to accomplish these steps are:

```
makedb -g eventdb
createdb eventdb
sch2db -D eventdb -y <VERSANT_ROOT>/lib/channel.sch
<VERSANT_ROOT>/lib/vedsechn.sch
```

Step 2: Start the Event Daemon

The event daemon is responsible for delivering the event notifications to the event clients, so it must be started before running the `StockWatcher` application. If you fail to start the event daemon, then event clients will fail when they try to perform channel administration tasks, such as creating a new channel.

The event daemon is named “`veddriver`,” and it accepts the name of a configuration file as a command-line argument. For the stock example, the event daemon should be started with the following command:

```
veddriver eventdb config.ved.sol
```

(If you are using Windows, replace “config.ved.sol” with “config.ved.win”). By default, this command runs in the “foreground,” so you should execute it and the subsequent commands in separate command windows.

Note that the configuration file contains the name of the database, so if you decide to use a different database, you should edit the configuration.

Step 3: Compile and Enhance

Before running the applications, they must be compiled and enhanced. If you forget the enhancement step, then the applications will get errors when trying to access the persistent StockQuote objects. The following commands will compile and enhance all of the Java source files:

```
javac *.java
java com.versant.Enhance .
```

Step 4: Run the Applications

After preparing the database and starting the event daemon, you can run the StockTrader and StockWatcher applications. You can start as many instances of these applications as you like, but you should create at least one StockTrader and one StockWatcher.

Running the StockTrader

The StockTrader application is not an event client, so it could actually be run before step 2. It does not depend on the event daemon. After starting the application, you should see a window that allows you to buy or sell shares of stocks.

```
java StockTrader eventdb
```

Note that the “eventdb” database will be populated with StockQuote objects automatically. Both the StockTrader and StockWatcher applications will create these initial StockQuote objects if they do not already exist in the database.

Running the StockWatcher

The StockWatcher is an event client, so starting it is a bit more complicated than starting the StockTrader. It must connect to the event daemon and listen for event notifications, so you must specify daemon and client hostname and port information. Assuming that the event daemon and event clients are running on the same machine, you can start the StockWatcher with the following command:

```
java StockWatcher eventdb localhost 4000 localhost 4001
```

Depending on your network configuration, you may need to replace “localhost” with the hostname for your machine.

The first “localhost” is the machine on which the event daemon is running. The number 4000 is the port on which the daemon is listening for connections from the event client. This port number is also specified in the event daemon configuration file.

The second “localhost” is the machine on which the event client (the StockWatcher) is running. The number 4001 is the port on which the event client will listen for connections from the event daemon. This must refer to an unused port on the client machine; otherwise, it is essentially an arbitrary number.

If you want to run another instance of the StockWatcher on the same client machine, then you need to select a different port number – for instance, 4002. Starting two event clients at the same hostname and port is an error. You can start a second instance of the StockWatcher with the following command:

```
java StockWatcher eventdb localhost 4000 localhost 4002
```

Using the StockTrader and StockWatcher Applications

After starting the StockTrader and StockWatcher applications, you can see how they interact. This is easy – just make some trades in the StockTrader window. For example, if you buy 1000 shares of VSNT stock, then the price will change from 20.6 to 21.6. This change should be reflected in all of the StockWatcher windows that are showing.

Even while making trades, you can start more instances of the StockTrader or StockWatcher applications. Just be sure to choose a unique port number for any additional event clients that you create.

Stopping the Applications

When you have finished experimenting with the stock example, you can exit the applications. Closing the windows will end the StockTrader and StockWatcher, but there is no interface for shutting down the event daemon. Your operating system should provide a simple means for stopping this process, such as the “kill” command, the Task Manager, or simply hitting CTRL-C. The event daemon will exit with an error if the database server is forcefully shut down.

Note that shutting down the `StockWatcher` event clients does not automatically remove the "stock" event channel. This channel will continue to exist, and any new event clients can access this channel to receive event notifications. Since event channels are persistent, you must explicitly delete them to remove them from the system. This can be done with the channel-management commands described in the following sections.

BUILDING JVI EVENT CLIENTS

This section explains how to use the JVI Event Notification API to build event clients and discusses some of the details and issues involved while using event notification.

All of the classes in the JVI Event Notification API are located in the package `com.versant.event`.

Basic Concepts

Types of Events

The events that can be generated in a Versant database are grouped into different categories. As you will see, these categories are reflected in the class hierarchy of the Event Notification API.

- **Class events** include the creation, modification and deletion of instances of a specified class.
- **Object events** include the modification and deletion of a given set of objects.
- **Transaction-marker events** indicate the beginning and end of a transaction.
- **User-defined events** are generated explicitly by a database client.

Types of Event Channels

Event channels are also categorized into different types. In addition, any channel can deliver notification of transaction-marker or user-defined events.

- **Class-based channels** deliver notification of class events for a specified set of classes.
- **Object-based channels** deliver notification of object events for a specified set of objects.

The EventClient Class

The `EventClient` class is the starting point for any JVI event client application. An `EventClient` object represents a connection to the event daemon. This class has three main responsibilities:

1. Encapsulating the event daemon connections.
2. Channel management functions.
3. Normal and exceptional event client termination

Connecting to the Event Daemon

All of the connection-related parameters are given in the `EventClient` constructor. The `StockWatcher` application essentially passes all of its command-line arguments to this constructor.

```
EventClient ( String daemonHost,
int daemonPort,
String clientHost,
int    clientPort,
String database    )
```

Note that the constructor does not immediately connect to the event daemon, so it will not fail if the daemon is not running.

Channel Management

After creating an `EventClient` object, you need to obtain one or more `EventChannel` objects in order to receive events. There are two ways to do this: access an existing channel by name, or define a new channel.

The `EventClient` object manages event channels on behalf of your application. It provides two methods for obtaining a new or existing channel:

```
EventChannel getChannel (String name) throws IOException
EventChannel newChannel
(String name, ChannelBuilder builder) throws IOException
```

The `StockWatcher` application tries to use the existing “stock” channel if it exists, and automatically creates the channel if it does not. The `getChannel` method returns `null` if there is no existing channel with that name, which is written as follows:

```
// If some other StockWatcher has already created an event
// channel named "stock", then get it.  Otherwise this call
// will return null.


EventChannel channel = client.getChannel ("stock");

// If the channel did not already exist, then create a new one.

if (channel == null) {
```

```
// The "stock" channel will be class-based, listening for
// changes to instances of the StockQuote class.

ClassChannelBuilder builder =
new ClassChannelBuilder ("StockQuote");

// Create a new channel using the channel builder.  After
// this, any event client can receive events on the "stock"
// channel.

channel = client.newChannel ("stock", builder);
}
```

NOTE:- There is a slight race condition in creating the “stock” channel. It is possible for two StockWatcher applications to be created simultaneously, so that the calls to getChannel in both programs return null. Then both applications would try to create a new channel, causing one of the two applications to fail with a ChannelException. One way to get around this problem would be to catch this ChannelException and retry the getChannel, which should succeed.]

Termination

The event client terminates normally when the client process terminates, or when the EventClient.shutdown method is invoked. After the event client terminates, the connection to the daemon is broken, and the daemon will no longer deliver event notifications to that client. The StockWatcher application terminates simply by calling System.exit.

Exceptional termination occurs when the connection with the event daemon is broken abnormally. For example, if the daemon is killed while clients are connected, these clients will terminate exceptionally. Since the EventClient is an “active” object, with the event processing and delivery occurring in a separate thread, the EventClient provides a mechanism – the ExceptionListener – for notifying applications of exceptions resulting in termination. The StockWatcher example uses this mechanism to print the stack trace and exit:

```
// If an exception occurs in the JVI event-handler thread, we
// can receive notification of this error by adding an
// ExceptionListener to the client.  This kind of exception
// typically occurs when the daemon quits abnormally, while
// there are still clients listening for events.

ExceptionListener exception_listener = new ExceptionListener ()
```

```
{
// When the event-handler thread calls exceptionOccurred,
// the EventClient automatically shuts down.  There is no
// no way to restart it; however, a new EventClient could
// be created.

public void exceptionOccurred (Throwable exception)
    {

// In this example, we simply print out the stack
// trace and exit.
exception.printStackTrace ();
        quit ();
    }
};
client.addExceptionListener (exception_listener);
```

The EventChannel Class

Once your application has an EventChannel object, it can listen for events that are delivered on this channel by adding event listener objects to the channel. This is the standard JavaBeans model for event notification, so `EventChannel` objects qualify as JavaBeans. The `EventChannel` class has two methods for managing listeners:

```
void addVersantEventListener    (VersantEventListener listener)
void removeVersantEventListener (VersantEventListener listener)
```

`VersantEventListener` is a simple interface that extends the standard JavaBeans `EventListener` interface. It does not define any methods. However, instead, subinterfaces of `VersantEventListener` separate the event functionality into different categories. These categories mirror the different types of events:

- **ClassEventListener** includes `instanceCreated`, `instanceModified` and `instanceDeleted` callbacks.
- **ObjectEventListener** include `objectModified` and `objectDeleted` callbacks.
- **TransactionMarkerEventListener** includes `beginTransaction` and `endTransaction` callbacks.
- **UserEventListener** contains the `userEvent` callback.

The StockWatcher application is interested in only one kind of event: modifications to instances of the StockQuote class. Therefore, the StockWatcher defines a single event listener that implements the instanceModified method. The listener is an anonymous inner class – a useful Java feature when building event-driven applications.

```
// A ClassEventListener is required to receive events on a
// class-based channel.
ClassEventListener event_listener = new ClassEventListener ()
{
    // We are interested only in modifications to StockQuote
    // objects.
    public void instanceModified (VersantEventObject event)
    {
        // ... process the event here ...
    }
public void instanceCreated (VersantEventObject event) {}
public void instanceDeleted (VersantEventObject event) {}

};

// Add the event listener object to the "stock" channel.  Now
// the instanceModified method will be invoked whenever a
// StockQuote object is modified.
channel.addVersantEventListener (event_listener);
```

In this example, the code that processes the event object is not shown; it is explained below in the section on the VersantEventObject class.

Note that immediately after calling addVersantEventListener, this event listener is “live” – it could start receiving event notifications right away. You need to make sure that your application is prepared to handle these event notifications.

The VersantEventObject Class

When an event notification is delivered to an event listener, the callback method is passed a VersantEventObject. This object contains all of the information about the event that occurred.

The StockTrader example uses the VersantEventObject to get the StockQuote object that was modified. This object is called the “raiser” object, since it is the object on which the event was raised. Only the identity of the object is delivered to the event client, so to obtain the new price of the modified StockQuote object, this object must be refreshed:

```
public void instanceModified (VersantEventObject event)
{
// Since the JVI event-handler thread is not the same
// as the main thread, we need to set the session of
// the thread before doing any database activity,
// including getting the persistent raiser object.

    session.setSession ();

// The raiser is the object that caused the event to
// be generated - in this case the StockQuote object
// that was modified.

    StockQuote quote = (StockQuote) event.getRaiser (session);

// The event tells us that the StockQuote object was
// modified in some other transaction.  This is only
// possible because the transaction associated with this
// session is not holding any locks on the object.
// Therefore, the instance in this session is "out-of-date."
// We can solve that by refreshing it.  We still don't
// want to keep the object locked, though, so we use
// NOLOCK.

Object[] array = { quote };
    session.refreshObjects

(array, session.database(), Constants.NOLOCK);

// Update the display of the symbol in the GUI panel.

updateSymbol (quote);

// We shouldn't leave the event-handler thread associated
// with this session.  If we did, then we could have

// trouble ending the session later on, because there would
// still be a thread associated with it.

    session.leaveSession ();
}
```

The ChannelBuilder Class

To create a new channel, you first construct a `ChannelBuilder` object. The `ChannelBuilder` allows you to specify which events will be generated on the channel. The `EventClient.newChannel` method has two arguments – the name of the new channel and the `ChannelBuilder` object that defines the parameters of the channel.

Corresponding to the three types of event channels, there are three subclasses of the abstract `ChannelBuilder` class:

- **ClassChannelBuilder** defines which classes will be monitored for a class-based channel. The `ClassChannelBuilder` has an associated set of classes. Event notifications are delivered on the channel when instances of these classes are created, modified or deleted.
- **ObjectChannelBuilder** defines which objects will be monitored for an object-based channel. The `ObjectChannelBuilder` has an associated set of persistent objects. Event notifications are delivered on the channel when any of these objects are modified or deleted.
- **QueryChannelBuilder** defines which objects will be monitored for a query-based channel. The `QueryChannelBuilder` has an associated set of query strings. Event notifications are delivered on the channel when an object satisfying any of these queries.

The `StockWatcher` application uses a `ClassChannelBuilder` to listen for changes to instances of the `StockQuote` class. In this example, only a single class is added to the `ClassChannelBuilder`, but additional classes could be specified with the `addClassName` method.

```
// The "stock" channel will be class-based, listening for
// changes to instances of the StockQuote class.

ClassChannelBuilder builder =
new ClassChannelBuilder ("StockQuote");

// Create a new channel using the channel builder.  After
// this, any event client can receive events on the "stock"
// channel.
channel = client.newChannel ("stock", builder);
```

NOTE:- A `ChannelBuilder` can only be used to create new channels; once created, the properties of a channel cannot be changed. However, you can delete an existing channel and replace it with a new one. Although you can not “ask” an `EventChannel` for a corresponding `ChannelBuilder`, the `ChannelBuilder` classes are serializable. This allows you to store the specification for an event channel so that it can be reused.

CHAPTER 7

Java Connector Architecture (JCA)

The J2EE Connector architecture provides a Java solution to the problem of connectivity between the many application servers and EISs already in existence.

The JVI JCA Connector is compliant with the JCA 1.0 specification. The JVI JCA resource adapters allow J2EE developers to enhance the performance of J2EE applications without having to learn an entire new API. The adapters that we have provided will allow you to manage your transactions in a variety of ways depending on the nature of your application and it's architecture. Furthermore, functions such as administration and management of connection pools will be managed by the Application Server.

This chapter describes the following topics:

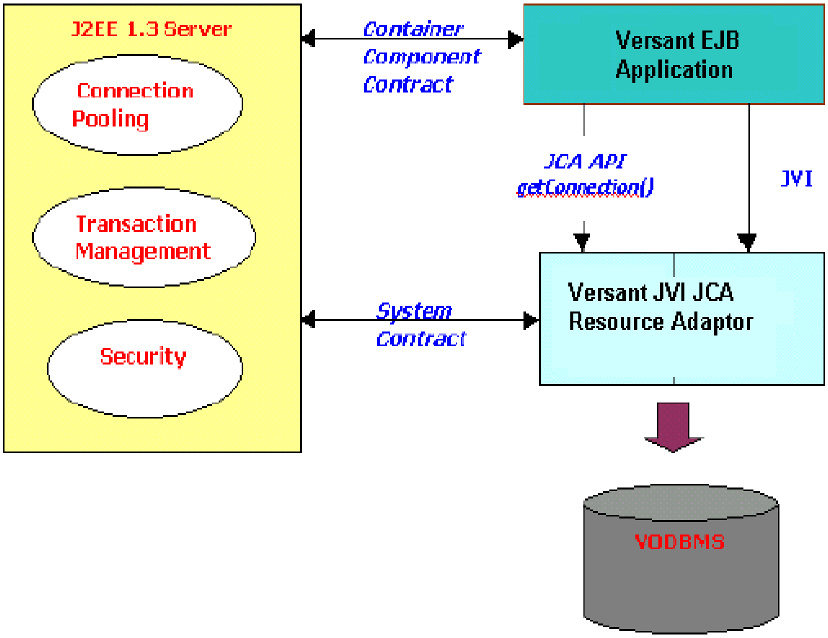
- The JVI J2EE Application Architecture
- Features
- Transactional Support
- Usage of JVI JCA Resource Adapters
- Deployment
- Non managed JCA
- JCA Error Constants

The JVI J2EE Application Architecture

In the connector architecture, Versant is a Resource Adapter provider. As a resource adapter provider, JVI provides a resource adapter that manages a set of shared Versant connections. Versant is a transactional resource manager, which means that it can participate in transactions. The transaction may either be coordinated by Versant RM itself (local transaction) or externally coordinated by a transaction manager (XA transaction mode).

The JVI J2EE application uses the standard JCA API to obtain a connection. After obtaining a connection, the application uses JVI to access the database.

The figure below illustrates the different components in a managed environment.:



In the above figure, the J2EE server, JVI JCA Resource Adapter and JVI are shown as separate entities. This is done to illustrate that there is a logical separation of the respective roles and responsibilities defined for the support of the system level contracts. However, this separation does not imply a physical separation, in terms of a container and resource adapter running in a separate process.

FEATURES

JVI JCA adapters have the following features:

Local Transaction Management

The JVI JCA Connector supports Local Transaction Management, thereby allowing the application server to manage resources which are local to the adapter. When the application requests for a connection, the application server starts a local transaction based on the transactional context.

Connection Sharing

Though this is an optional feature in JCA, it is supported in the JVI JCA Connector. When multiple connections acquired by a J2EE application using the same resource manager, containers may attempt to share connections within the same transaction scope. Sharing connections typically results in efficient usage of resources and better performance. In simple terms, connection-sharing support means that multiple `getConnection` requests on the same resource manager in the same transaction return the same connection.

Common Client Interface Support

The CCI defines a standard client API for application components. The CCI enables application components and EAI frameworks to drive interactions across heterogeneous EISs using a common client API. It is targeted primarily towards application development tools.

The CCI interfaces can be divided into four categories which are Connection-related interfaces, Interaction related interfaces, Data-representation-interfaces and Metadata-related interfaces.

The JVI JCA Connector will support the Connection-related and Metadata-related interfaces.

TRANSACTIONAL SUPPORT

JVI provides adapters that supports all three types of transactions specified in the JCA specification and include the following:

No Transaction

In this instance, the adapter supports neither local nor JTA transactions. It does not implement either XAResource or LocalTransaction interfaces.

The configurable properties of this adapter may be modified by making any applicable changes to the following file located at `<VERSANT_ROOT>\jvi\jca\vodNOTX.rar`.

Local Transaction

In this instance, the adapter supports resource manager local transactions by implementing the LocalTransaction interface as specified in the JCA specification. Typically, these transactions are demarcated by the container, but should you wish to demarcate these transactions, you may use cci.LocalTransaction to do so.

The configurable properties of this adapter may be modified by making any applicable changes to the following file located at `<VERSANT_ROOT>\jvi\jca\vodLOCAL.rar.`

XA Transaction

This adapter supports JTA transactions. As was the case with local transactions, these transactions are demarcated by the container typically, but should you wish to demarcate these transactions, you may use the JTA API to do so.

The configurable properties of this adapter may be modified making any applicable changes to the following file located at: `<VERSANT_ROOT>\jvi\jca\vodXA.rar`.

NOTE:- Although the JCA specification does allow support for local transactions, JVI's XA adapter cannot be used for local transactions.

USAGE OF JVI JCA RESOURCE ADAPTERS

As a J2EE developer, you will notice that utilizing JVI resource adapters is no different than what you would typically do in application that does not use JVI.

There are three sequential steps involved in usage of these adapters as given below, which are irrespective of the nature of your transactions.

Getting a connection factory

Look up the connection factory based on the JNDI name configured in `weblogic-ra.xml`. The JNDI name passed in the method `NamingContext.lookup` is the same as that specified in `jndi-name` element of the deployment descriptor (weblogic-ra.xml).

The JNDI look up results in a connection factory instance of the type

`javax.resource.cci.ConnectionFactory` as specified in the `connectionfactoryinterface element of the` deployment descriptor (ra.xml).

```
//obtain the initial JNDI Naming Context

Context initcnxt = new InitialContext();

//perform the JNDI lookup to obtain the connection factory

javax.resource.cci.ConnectionFactory cfy =

(javax.resource.cci.ConnectionFactory)initcnxt.lookup

(“java:comp/env/eis/MyEIS”);
```

Specific implementation is described in detail within each individual section as it differs on which component of the J2EE specification you use in your application.

Getting a connection

Once you have looked up the connection factory, all you have to do is get a connection within each method that uses Transparent JVI. The `getConnection()` call establishes a connection to an Versant database instance. The returned connection instance represents an application-level handle to an underlying physical connection.

The snippet of code below represents how a connection handle is obtained:

```
javax.resource.cci.Connection cxn = cfy.getConnection();
```

```
TransSession session = ((VConnection)cxn).getTransSession();
```

Once you have a connection, you can retrieve a session and interact with the Versant Object database using transparent JVI as shown in the code above.

NOTE:- VConnection is an extension of the JCA CCI Connection interface, which provides methods for getting a handle to JVI's TransSession.

IMPORTANT NOTE FOR XA ADAPTER USAGE:- If you are using the JVI's XA adapter, the getConnection() can only be made within the context of a global transaction and must be called only after the transaction has commenced.

Closing the connection

Once you have completed the necessary operations, all you have to do is close the connection by using the close method on the Connection interface as shown below:

```
cxn.close();
```

DEPLOYMENT

Deployment of JVI resource adapters is similar to deployment of EJBs and Enterprise Applications. We highly recommend deploying the resource adapter through the WebLogic Administration Console.

WARNING:- However if you do choose to do it from the command line, please make a back up copy because those files are critical for running the JCA demos packaged with JVI.

JVI Resource adapters use two deployment descriptors to define the operational parameters – `ra.xml` as defined by the JCA specification and the application server specific descriptor, `in` this case `weblogic-ra.xml`, which defines parameters unique to WebLogic.

The subsequent part of this section is dedicated primarily to the properties that the developer can customize and configure in ra.xml, which is specific to the JVI resource adapters.

The beginning of the deployment descriptor as shown below is in compliance with the JCA specification for deployment of resource adapters. This part does not require any modifications and essentially provides general information about Versant.

```
<display-name>vodXA</display-name>
<vendor-name>Versant Corporation</vendor-name>
<spec-version>1.0</spec-version>

<eis-type>Versant ODBMS</eis-type>
<version>1.0</version>
<resourceadapter>
<managedconnectionfactoryclass>
com.versant.connector.jvi.spi.JviXATxManagedConnectionFactoryImpl
</managedconnectionfactory-class>
<connectionfactory-interface>
javax.resource.cci.ConnectionFactory
</connectionfactory-interface>
<connectionfactory-impl-class>
com.versant.connector.cci.ConnectionFactoryImpl
</connectionfactory-impl-class>
<connection-interface>
javax.resource.cci.Connection
</connection-interface>
<connection-impl-class>
com.versant.connector.jvi.cci.VConnectionImpl
```

```
</connection-impl-class>
<transaction-support>XATransaction</transaction-support>
```

In the latter part of the descriptor, the JVI JCA Connector provides the following configurable properties based on which, the Connection Factory creates connections to Versant's underlying object database.

ConnectionURL

This element allows the user to specify the name of the Versant Database.

Usage is shown below:

Name	ConnectionURL
Type	java.lang.String
Value	Single Value
Max Length restriction	31 characters for the database name
Example Value	voddb

Example of usage:

```
<config-property-name>ConnectionURL</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>voddb</config-property-value>
```

Session Type

This element allows the user to define the Session type.

Usage is shown below.

Name	SessionType
Type	java.lang.String
Value	Single Value
Default Value	TRANSSESSION

Name	SessionType
Max Length restriction from TranSession	Nil
Example Value	TRANSSESSION, FUNDSESSION, SHARED_SESSION, or a fully qualified classname representing the custom implementation. The custom class should subclass com.versant.fund.FundSession and define a constructor with java.util.Properties argument.

NOTE:- SHARED_SESSION option is valid only for adapters with LocalTransaction support and NoTransaction support. Shared Session is a special type of connection, which should be used for read-only operations. This type of connection should be used only by users who are familiar with JVI. When a user specifies a SHARED_SESSION type, then all the managed connections use the same underlying connection. This session reads objects with DROP_RLOCK mode. Commit is disabled for this connection type. However, when it reaches the zap-threshold and when no user is using the connection, then a clean COD operation will be done. Since, the session does not retain locks, hence it is possible that it caches stale objects.

Advanced Usage Value: com.session.MySession (The name of the class provided by the user representing the custom implementation of Session as described above.)

Example of usage:

```
<config-property-name>SessionType</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>TRANSSESSION</config-property-value>
```

Connected Databases

This element allows the user to specify the database/s to which connection/s is/are desired in addition to the session database specified with the ConnectionURL.

Please ensure that the session database is not listed in this element.

This property is valid in the case of Local Transactions and No Transaction.

Usage is as follows.

Name	ConnectedDatabases
Type	java.lang.String
Value	Space Separated database names
Max Length restriction from TranSession	31 characters
Example Value	db1 db2 db3

Example of usage:

```
<config-property-name>ConnectedDatabases</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>db1 db2</config-property-value>
```

XA BranchName

This element is used for branch names. Internally, the Resource Adapter will create unique ID's for each session with the help of this prefix. This property is only valid for XA Transactions

Name	XABranchName
Type	java.lang.String
Value	Single value
Max Length restriction from TranSession	31 characters
Example Value	Session1

Example of usage:

```
<config-property-name>XABranchName</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>XaSession1</config-property-value>
```

SessionOptions

This element allows the user to specify the Session Options. These are;

DEFAULT: Start a "standard session" that uses the standard VERSANT session, transaction, and locking models. This is the default.

OPTIMISTIC: Start an "optimistic locking session" that suppresses object swapping, prevents automatic lock upgrades, and provides automatic collision notification.

READ: Start a session with read connection mode that allows you to view but not change data. If you make a connection with a read mode access and then change an object, you will get the error NET_RW_ACCESS_REQUIRED when you try to commit your transaction (permissions are checked by the database server process).

READ_WRITE: Start a session with connection mode that allows you to view and edit data in your session database.

NOTE:- If you are using the SHARED_SESSION Type as described earlier, this property is configured to be READ by default and the only valid value for SessionOptions is READ.

Usage is as follows:

Name	SessionOptions
Type	java.lang.String
Value	Separate the following values with comma
Default Value	DEFAULT
Max Length restriction from TranSession	Nil
Example Value	READ_WRITE

Example of usage:

```
<config-property-name>SessionOptions</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value> DEFAULT</config-property-value>
```

LockMode

This element allows the user to specify the lock mode. Although the default lock mode is RLOCK, the following can be specified in this element.

NOLOCK	Use no lock as the default lock mode for this session.
WLOCK	Use write lock as the default lock mode for this session.
ULOCK	Use update lock as the default lock mode for this session.
RLOCK	Use read lock as the default lock mode for this session.

Usage is as follows:

Name	LockMode
Use	The default lock mode for the session. Default is RLOCK. For SessionType as SHARED_SESSION, the only valid value for LockMode is RLOCK.
Type	java.lang.String
Value	Any of the Following
Max Length restriction from TranSession	Nil
Default Value	RLOCK

Example of usage:

```
<config-property-name>LockMode</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value> RLOCK </config-property-value>
```

UserName

This element allows specifying a valid database user. If the userName being specified is the name of the DBA, this user must also be the OS user.

Usage is as follows.

Name	UserName
Type	java.lang.String
Value	Single Value
Max Length restriction from TranSession	Max Length 31 Characters
Example Value	dbuser

Example of usage:

```
<config-property-name>UserName</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>dbuser</config-property-value>
```

Password

This element allows for providing a password for the user specified in the userName property described above. If the UserName has not been set previously, then this property need not be set.

Usage is as follows:

Name	Password
Type	java.lang.String
Value	Single Value
Max Length restriction from TranSession	Max Length 256 Characters
Example value	dbpassword

Example of usage:

```
<config-property-name>Password</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>dbpassword</config-property-value>
```

EvolveSchema

This element allows enabling of schema evolution.

Usage is as follows:

Name	EvolveSchema
Type	java.lang.Boolean
Value	true or false
Default Value	false

Example of usage:

```
<config-property-name>EvolveSchema</config-property-name>
<config-property-type>java.lang.Boolean</config-property-type>
<config-property-value>true</config-property-value>
```

ZapThreshold

This element allows the user to configure the policy for cleaning up the COD table. The policy can be based either on the number of transactions or the time interval. The cleaning of the COD is done only after exceeding the zap threshold and when the connection is not being used.

Usage is as follows:

Name	ZapThreshold
Type	java.lang.String
Value	A name-value pair consisting of the policy name and it's value
Example Values	1. NumTx=10 (where NumTx is the chosen policy based on num of transactions and 10 is the value) 2. ZapTimeout=600 seconds(where ZapTimeout is the chosen policy based on the time interval between the flush operations and 600 is the value in milliseconds)
Default Value	ZapTimeout=600 seconds

Example of usage:

```
<config-property-name>ZapThreshold</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>NumTx=10</config-property-value>
```

LogString

This option allows the user to enable the logging filter or tracing level. The default value for the element is OFF.

Usage is as follows:

Name	LogString
Type	java.lang.String

Name	LogString
Possible Values	OFF, DEBUG, VERBOSE
Default Value	LOGLEVEL=OFF

Example of usage:

```
<config-property-name>LogLevel</config-property-name>
<config-property-type>java.lang.String</config-property-type>
<config-property-value>LOGLEVEL=OFF</config-property-value>
```

NON MANAGED JCA

The JVI JCA Connector provides a ConnectionManager implementation that will do the pool management for non-managed scenarios as may be in the case for applets or Java applications.

In a non-managed environment, the Versant client will not use any of the JVI JCA Connectors resource adapters. The user will configure `javax.resource.spi.ManagedConnectionFactory` instance programmatically and can be done in either of the following ways:

Setting the individual properties using the mutators provided by `ManagedConnectionFactoryImpl class.`

For more details of the //omitted code try

```
{
ManagedConnectionFactoryImpl mcImpl=new ManagedConnectionFactory
Impl();
mcImpl.setDatabaseName (dbname);
conFact = (javax.resource.cci.ConnectionFactory)
mcImpl.createConnectionFactory ();

//omitted code
providing a Object containing the name-value pairs as an input to
the ManagedConnectionFactoryImpl class
```

NOTE:- For non-managed applications, please refer to javadoc of `com.versant.connector.jvi.nonmanaged.ManagedConnectionFactoryImpl`

JCA ERROR CONSTANTS

Error Number	ERROR	ERROR DESCRIPTION
14001	EJC_INVALID_URL	ConnectionURL cannot be null or empty or exceed 31 chars in length
14002	EJC_INVALID_LOCKMODE	Valid values for session lockmode are NOLOCK, WLOCK, ULOCK and RLOCK
14003	EJC_INVALID_ZAP_THRESHOLD	Invalid configuration for zap threshold policy
14005	EJC_INVALID_LOG_LEVEL	Valid values for LogLevel are OFF, DEBUG or VERBOSE only
14006	EJC_INVALID_CONFIG_MAN	ConfigManagerName cannot be an empty string and should implement `com.versant.connector.common.ConfigManager interface.`
14007	EJC_INVALID_SESSION_TYPE	Valid values for SessionType are FUNDSESSION, TRANSSESSION, SHARED_SESSION or a classname representing the custom implementation of FundSession. **NOTE:-** SHARED_SESSION is not a valid option for adapter with XATransaction support.
14008	EJC_INVALID_OPTIONS	Valid values for SessionOptions are the comma separated combination of DEFAULT, OPTIMISTIC, READ, READ_WRITE
14009	EJC_INVALID_USERNAME	UserName length cannot be more than 31 chars

Error Number	ERROR	ERROR DESCRIPTION
14001	EJC_INVALID_URL	ConnectionURL cannot be null or empty or exceed 31 chars in length
14010	EJC_INVALID_PASSWORD	Password length cannot be more than 256 chars
14011	EJC_INVALID_CONNDBS	Invalid configuration for connected databases
14012	EJC_INVALID_BRANCH_NAME	Invalid configuration for XA BranchName, cannot be empty or greater than 31 chars in length"
14013	EJC_NOT_SUPPORTED	Operation is not supported
14014	EJC_DIFF_SESSTYPE	Session type is different from TransSession
14015	EJC_ILLEGAL_ARGTYPE	Illegal argument

CHAPTER 8

Versant JTA/XA

Versant JTA/XA supports managed and non-managed environment.

This chapter describes how Versant supports the X/Open XA protocol as a resource manager in Java, based on the XA interface of the Java Transaction API package, version 1.0.

Following topics are covered:

- Versant JTA - XA in non-managed environment
- Interfaces and Classes in the com.versant.xa package
- VXAResource
- VSessionPool
- Limitations
- Demos
- Versant JTA in managed environment
- XA Recovery

VERSANT JTA/XA SUPPORT IN A NON-MANAGED ENVIRONMENT

Sun Microsystems, Inc. has released the Java Transaction API specification, which consists of a high-level application transaction interface and a Java mapping of the industry-standard X/ Open XA interface. The application transaction interface allows an application to control user transaction boundaries.

The XA interface allows an external transaction manager to control transaction boundaries for operations performed by multiple resource managers using the two-phase commit X/Open XA protocol. This chapter describes how Versant supports the X/Open XA protocol as a resource manager in Java, based on the XA interface of the Java Transaction API package, version 1.0.

Transaction processing work in a distributed system requires the co-operation of three different parties:

- The Transaction Manager (TM) which assigns identifiers to transactions, monitors their progress and takes responsibility for transaction completion and recovery.
- The Resource Manager (RM) that provides access to shared resources within a transactional context.
- The Application Program (AP) which requires transaction services to perform useful work.

In the XA specification terminology, JVI and the Versant ODBMS, function as a Resource Manager. By implementing the standard `javax.transaction.xa.XAResource` interface, JVI can participate in a global, distributed transaction that is coordinated by an external Transaction Manager. Within an externally managed transaction, an Application Program can use a Versant session to create and access persistent objects, perform queries, etc. – the same capabilities that are available to applications that don’t use the JTA/XA protocol.

Since Versant is an object-oriented database system, it does not implement the JDBC-specific interfaces such as `javax.sql.XADataSource` and `javax.sql.XAConnection`. These interfaces allow connection pooling to be performed by the Application Server. In contrast, Versant JTA/XA implements only the standard `javax.transaction.xa.XAResource` interface, and performs connection pooling within this framework.

All of the Versant JTA/XA interfaces and classes can be found in the `com.versant.xa package`.

Interfaces and Classes in the com.versant.xa Package

- **SessionFactory** - Instances of a class that implement this interface are used to create sessions
- **SessionPool** - This interface provides methods to get and release sessions from the session pool. Any Session pool implementation that is used with the `VXAResource` class must implement this interface.
- **TMFlags** - This class provides constant definitions for all the Transaction Manager flags.
- **VSessionPool** - Default implementation of the `SessionPool` interface.
- **VXAResource** - Implements the `javax.transaction.xa.XAResource` interface.
- **VXid** - Implements the `javax.transaction.xa.Xid` interface.

VXAResource

The `com.versant.xa.VXAResource` class implements the `javax.transaction.xa.XAResource` interface. The TM obtains a `XAResource` for each RM participating in a global transaction. Only one `VXAResource` per resource is needed to handle all of the application's transactional work. This is because each `VXAResource` instance explicitly manages the connections to the database. Towards this end, session pooling is done within the framework of the class itself. When a connection to the database is needed, any one of the unused connections is acquired from the session pool and the appropriate task is accomplished.

Invoking the start method on the `VXAResource` instance starts a transaction. Once a connection to the database for beginning the transaction is acquired from the session pool, the connection is associated with the `Xid` that identifies this transaction. This connection in the session pool cannot be acquired by any other transaction until the transaction is disassociated with the connection by calling `VXAResource.end`.

After a transaction has started via `VXAResource.start`, the Application Program will require a session to do useful work. The Application Server can provide the session mapped to a `Xid` to the Application Program by using the `SessionPool.get(Xid)` method. The `VXAResource.getSessionPool()` method returns the SessionPool associated with the `VXAResource`.

A `VXAResource` instance can be constructed using either of the two constructors. The simplest way to construct a `VXAResource` instance is to use the constructor that takes the database name as a parameter. This constructor in addition to initializing the `VXAResource` instance provides a default implementation for the `SessionPool`. The `SessionPool` interface provides methods to acquire a connection that is mapped to a particular `Xid` as well as to a connection that is unmapped. The SessionPool thus created creates one initial session and allows INTEGER.MAX_VALUE number of sessions to be created.

The simple form of the constructor does not allow the user to customize the session pool or the sessions created in the session pool. If the user required FundSessions to be created in the pool or to pass special options to the session constructor, it is not possible using this form of the constructor. For users requiring customized SessionPools, the other form of the VXAResource constructor needs to be used. Details regarding the usage of this constructor are provided in the VSessionPool section below.

VSessionPool

The com.versant.xa.VSessionPool class provides a default implementation of the com.versant.xa.SessionPool Interface.

The SessionPool Interface provides the following three methods:

1. FundSession get()
2. FundSession get(Xid)
3. void release(FundSession)

The get method returns an unmapped connection from the pool. A connection is said to be unmapped if no transaction is associated with this connection. To get a session that is mapped to a particular transaction, the get(Xid) method can be used. The release method releases the connection back to the pool. It is very important that every connection that is acquired using the get() method be released back to the pool. If the user is using the VSessionPool class, not releasing the connection will mean that the session pool capacity reduces by one.

The properties of the session object returned by the get methods in the VSessionPool instance depend on the implementation of the create method in the SessionFactory interface. A reference implementation of the SessionFactory interface is shown below:

```
SessionFactory sessionfactory = new SessionFactory() {
  public FundSession create()
  {
    Properties p = new Properties();
    p.put("database", dbName);

  // These options are necessary for any session to be used in a XA
    // Transaction. Any other options that need to be passed on to the
    // session constructor can be added here.

    p.put("options", (Constants.DONT_JOIN|Constants.EXTERNAL_XACT)+"");

    // When EXTERNAL_XACT is used as an option, then the session name
    // has to be provided.  A default name will not be provided.

    p.put("sessionName", "session" + sessionCount++);
    return new FundSession(p);
  }
```

Limitations

Versant JTA/XA has the following limitations:

1. Dynamic registration is not supported. The Versant RM does not support the `ax_reg` call that would allow dynamic registration of the resource manager with the transaction manager.
2. Transaction time out is not supported. The user cannot use the `VXAResource.setTransactionTimeout` to set the Transaction time out. Consequently any call to this API will always return false. Due to the same limitation `getTransactionTimeout` always returns 0.
3. `VXAResource.start` cannot be invoked with the `TMJOIN` flag. This means that `VXAResource.start` cannot be used to join an existing transaction branch.
4. When a Transaction is suspended, the session is not released back into the pool. The session is only released back into the pool when `VXAResource.end` is called with either `TMSUCCESS` or `TMFAIL` flag. If too many transactions are suspended then the session pool capacity can be seriously affected.
5. Thread migration works only with threads that share the same `VXAResource` instance. If `VXAResource.end` was called by thread ‘A’ with the `TMSUSPEND` flag, then thread ‘B’ can resume the transaction by calling `VXAResource.start` with the `TMRESUME` flag only if the same `VXAResource` instance is used.

Demos

There are two usage demos provided with your JVI installation.

The XADemo1 demonstrates Versant JTA/XA usage with `TransSession` while XADemo2 demonstrates usage with `FundSession`. For a proper understanding of the demos the user should be familiar with the X/Open CAE Specification.

Both of the demos are very simple Versant applications. There are multiple transactions within each demo. Work done within each transaction is delineated by the `VXAResource.start` and `VXAResource.end` calls. After `VXAResource.end` has been called, the transaction is either committed or rolled back.

The XADemo2 demo uses a Customized session pool. The SessionFactory interface is implemented as an inner class and creates FundSession instances. In addition this demo demonstrates how to suspend and resume a transaction.

Running the Demos

To start with, create a Versant database that will be used by this application. `xadb` is used as the database name here.

The commands to accomplish this are:

```
% makedb -g xadb
% createdb xadb
```

The `config.jvi` file is already setup for you. The `config.jvi` declares class ‘Person’ to be Persistent Capable. Instances of only this class will be stored in the database. Compile the java files. Enhance the class files by running the enhancer. The commands to compile and enhance are:

```
% javac XADemo1.java XADemo2.java
% java com.versant.Enhance .
```

We are now ready to run the demos. Run the demo by typing the command:

```
 %java XADemo[1or2] xadb
```

VERSANT JTA SUPPORT IN A MANAGED ENVIRONMENT

JTA is a service-oriented API specification. Most of the J2EE application servers support JTA. This feature allows the application server to perform distributed transactions over multiple heterogeneous datasources.

The application servers contain a Transaction Manager, that coordinates distributed transactions with any XAcompliant resource. JTA support in Versant allows users to perform distributed transactions using the two phase commit protocol between Versant and other JTA compliant data sources such as an RDBMS or a JMS queue.

The JVI JCA Connector provides a Versant XA Resource Manager that implements the JTA XAResource interface. Using JTA with the JVI JCA Connector does not involve a high learning curve. As a J2EE programmer, you will find that you can reuse all of your prior knowledge.

There are essentially four steps that are required to implement JTA using the JVI JCA Connector.

Step 1: Start a global transaction

This is the first step in JTA usage with the JVI JCA Connector. If your transactions are container demarcated, you may skip this step, as the transactions are started by the container before method execution.

In the code snippet below, the transaction has been user demarcated and reflects starting of the global transaction.

```
public void beanTransactionalMethod ()
{
Context ctx = new InitialContext ();
// First obtain UserTransaction
UserTransaction utx = (javax.transaction.UserTransaction)
ctx.lookup("javax.transaction.UserTransaction");
//omitted code
try {
Global transaction
utx.begin ();
//
```

Step 2: Get a Connection

Once you have started a global transaction, you can get a connection as shown below.

The code snippet below demonstrates a connection to an Versant database instance. Once you have a connection you can retrieve a session and interact with the Versant object database using the transparent JVI as shown below:

NOTE:- `VConnection` is an extension of the JCA CCI Connection interface, which provides methods for getting a handle to JVI's TransSession.

```
//Obtain connection to JVI JCA Connector
javax.resource.cci.ConnectionFactory conFact =
(javax.resource.cci.ConnectionFactory)ctx.lookup ("eis/vodJNDIXA");
vconn = (VConnection) conFact.getConnection();
TransSession session = vconn.getTransSession ();
//Do database operations in Versant
```

Step 3: Close the Connection

Once you have completed the necessary operations, you can close the connections as shown below:

```
vconn.close ();
```

Step 4: End the global transaction

Again, this step is necessary only in the event that your transactions are NOT container demarcated.

The snippet of code below shows ending the global transaction in a user demarcated transaction.

```
utx.commit ();
```

XA Recovery

There are two key components to Failure Recovery with the JVI JCA Connector.

1. Before starting your Application Server, please ensure that your Init Pool size is set to greater than zero. This ensures recovery after initialization time.
2. If the Application Server crashes, then the user must start the Versant object Database. Performing this step, rollsback transactions that have not reached the prepared phase.

NOTE:- Starting the application till the XA recovery is complete is not recommended. To determine if there are any pending transactions, please check the locks in the database. If you do notice that there are locks present, it indicates, that the process of recovery is not complete.

CHAPTER 9

Session Managed Persistence

This approach is the most powerful of the design choices.

It gives the developer the flexibility to use the JVI JCA Connector's transparent object management capabilities, while allowing the Application Server to manage connection pooling and transactions.

This way the developer only has to focus on development of the business logic.

This chapter covers the following in detail:

- Introduction to SMP
- Developing the Bean
- Specify the Bean deployment descriptor
- Deploying the JVI JCA Resource adapter
- Building and Deploying an SMP Bean

INTRODUCTION TO SMP

You can use easily use the transparent JVI from a Session Bean to store and retrieve objects persistently. The data objects in your application can be directly persisted by Versant database without any need for mapping.

You create the Java classes that represent your persistent data objects, and develop Session Beans using transparent JVI to access the data objects. JVI uses JCA adapters to tie in with the application server’s connection pooling and transaction management capabilities.

DEVELOPING THE BEAN

To demonstrate use of the Session Beans, we will use a `BankBean` as an example.

The purpose of the sections that follow is primarily to assist the developer in usage of JVI JCA Connector in your application. Snippets of code will be provided as necessary.

All the files for the example and the complete code for the example being used can be found in `<VERSANT_ROOT>\jvi\demo\jca\smp`.

Irrespective of the transaction type you choose to use, there are three steps that are needed to get started:

- Getting a connection factory
- Getting a connection
- Closing a connection

Getting a connection factory

In the setSessionContext() method, look up the connection factory based on the JNDI name configured in weblogic-ra.xml. In the example that we have provided, this is done as follows:

```
public void setSessionContext (SessionContext sc) {
this.sc = sc;
StringBuffer buf = new StringBuffer();
try {
buf = new StringBuffer();
InitialContext ctx = new InitialContext();
String connectionName =
(String)ctx.lookup("java:comp/env/connection-factory-jndi-name");
conFactory = (ConnectionFactory)ctx.lookup (connectionName);
//omitted code
}
```

Although it is not mandatory to look up the connection factory in the `setSessionContext`() method, we recommend this approach for efficiency.

Getting a connection

Once you have looked up the connection factory all you have to do is get a connection within each business method that uses transparent JVI. The `getConnection()` call associates an existing connection or creates one to an instance of Versant database, and returns the corresponding application level connection. In the snippet of code below, all we have done is get a connection.

```
public String findMember (long ssn) {
Connection connection = null;
try {
connection = conFactory.getConnection();
//omitted code
Once you have the connection, you
can retrieve the session as shown below from the same connection, and
then interact with Versant object database as shown
below (in the case of the example, a query is used to find a bank mem-
//omitted code
TransSession session = ((VConnection)connection).getTransSession();
//omitted code
```

Closing the connection

Once you have completed the necessary operations, all you have to do is close the connection by calling the `closeConnection()`method in the finally block of each business method as shown below:

```
//omitted code
finally {
closeConnection (connection);
}
//omitted code
The above code closes the application connection handle and the under
lying physical connection is released back to the connection pool.

Implementation of the closeConnection() method in our Session Bean
example is shown below.

private void closeConnection(Connection conn) {
try {
```

```
if (conn != null) {
conn.close();
}
} catch (ResourceException re) {
throw new EJBException ("ResourceException in closing connection "
+ re.getMessage(), re);
}
}
```

SPECIFY THE BEAN DEPLOYMENT DESCRIPTOR

In order to deploy a `SessionBean`, the deployer needs to create the SessionBean's deployment descriptor `ejb-jar.xml` file.

Please see your Application Server documentation for details.

The example below demonstrates the ejb-jar.xml file for the example we have been building in this section.

You will notice, that the component of this that deviates from developing a Session Bean without Versant connector specific details is the `<env-entry>` which is used to specify the details of the connection factory.

```
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>SMP_Bank</ejb-name>
<home>smp.stateless.BankHome</home>
<remote>smp.stateless.Bank</remote>
<ejb-class>smp.stateless.BankBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
<env-entry>
<env-entry-name>connection-factory-jndi-name</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>eis/vodJNDILocal</env-entry-value>
</env-entry>
</session>
</enterprise-beans>
<assembly-descriptor>
<container-transaction>
<method>
<ejb-name>SMP_Bank</ejb-name>
<method-intf>Remote</method-intf>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
```

DEPLOYING THE JVI JCA RESOURCE ADAPTER

Prior to deploying, the user must ensure that they have created an instance of an Versant database. You can create a database by simply using the following:

```
makedb <dbName>
createdb <dbName>
```

where <dbname> will be the name for the Versant database instance – for example, voddb

For more information on Creating databases, please refer to the Database Administration Manual, Chapter 1: Creating a Database.

Following are the steps to deploy the JVI JCA Resource Adapter to WebLogic 9.0, Websphere 6.1 and JBoss 4.0

Deploying to WebLogic 9.0

1. Add the jar dependency `$VERSANT_ROOT/lib/jvi7.0.1-jdk1.4.jar`.

   You can copy this file in the lib directory of the domain.

   ```
   ($BEA_HOME/user_projects/domains/$DOMAIN_NAME/lib)
   ```

   or make sure that `jvi7.0.1-jdk1.4.jar` is in the `CLASSPATH` before the WebLogic server is started.

2. You can configure the database specific information and other details in `ra.xml` of the JCA adapter rar file before you deploy the adapter.

   For example:

   ```
   <config-property>
      <config-property-name>ConnectionURL</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-value>voddb</config-property-value>
    </config-property>
    <config-property>
      <config-property-name>SessionOptions</config-property-name>
      <config-property-type>java.lang.String</config-property-type>
      <config-property-value>DEFAULT</config-property-value>
    </config-property>
   ```

3. Start the WebLogic server using

   ```
   ${WL_USER_PROJECTS}/${WL_DOMAIN}/startWebLogic.cmd
   or
   ${WL_USER_PROJECTS}/${WL_DOMAIN}/startWebLogic.sh
   ```

4. Deploy the JCA adapter rar file.

   You can copy the appropriate JVI JCA adapter rar file as below, in the autodeploy folder of your Weblogic domain
   `$BEA_HOME/user_projects/domains/$DOMAIN_NAME/autodeploy`

   Files are:

   ```
   $VERSANT_ROOT/jvi/jca/vodLOCAL.rar,
   $VERSANT_ROOT/jvi/jca/vodNOTX.rar,
   $VERSANT_ROOT/jvi/jca/vodXA.rar
   ```

   or you can deploy the rar file using the domain’s admin server console.

Deploying to WebSphere 6.1

Steps to Deploy the JVI JCA resource adapter to WebSphere 6.1

1. Make sure that WAS (WebSphere Application Server) is running.

   To start the WebSphere 6.1 server you can follow the "Start Server" link option from

   Start -> Programs -> IBM WebSphere -> Application Server V 6.1 -> Profiles -> AppSrv01 -> FirstSteps

   or use the command line option `{WAS_HOME}/bin/startServer.bat <server>`

   or `{WAS_HOME}/bin/startServer.sh <server>`

2. Open the Admin console of WAS in a web browser and add a Shared Library as follows:

   Go to Environment -> Shared Libraries -> Scope - select the Node and Server on which the Shared Library needs to be installed

   In *Preferences -> Click New*

   Add the following details and Apply and Save the changes:

   ```
   Name: JVIJCA
   Classpath: <VERSANT_ROOT>/lib/jvi7.0.1-jdk1.4.jar
   Native Library Path: <VERSANT_ROOT>/bin
   ```

3. Using the admin console install the JVI JCA local resource adapter as follows:

   Go to Resources -> Resource Adapters -> Resource adapters -> Scope - Select the Node and Server on which the resource adapter needs to be installed

   Click on the Install RAR button

   Add the following details, Apply and Save the changes:

   Local path (to the rar file): select the file `<VERSANT_ROOT>/jvi/jca/vodLOCAL.rar`

   ```
   Name: vodLOCAL
   Classpath: <VERSANT_ROOT>/lib/jvi7.0.1-jdk1.4.jar
   ```

   Once the Resource adapter is created, install a J2C connection factory over the vodLocal resource adapter created as follows:

   Click on J2C connection factory

   Add the following details, Apply and Save the changes.

   ```
   Name: vodLocalConnectionFactory
   Jndi name: eis/vodJNDILocal
   ```

Deploying to JBoss 4.0.x

1. Add the jar dependency `$VERSANT_ROOT/lib/jvi7.0.1-jdk1.4.jar`.

   You can copy this file in the lib directory of the domain

   `($JBOSS_HOME/server/default/lib}`

   or make sure that `jvi7.0.1-jdk1.4.jar` is in the `CLASSPATH` before the JBoss server is started.
2. You can configure the database specific information and other details in `ra.xml` of the JCA adapter rar file before you deploy the adapter.

   For example:

```
<config-property>
   <config-property-name>ConnectionURL</config-property-name>
   <config-property-type>java.lang.String</config-property-type>
   <config-property-value>voddb</config-property-value>
 </config-property>
 <config-property>
   <config-property-name>SessionOptions</config-property-name>
```

```
    <config-property-type>java.lang.String</config-property-type>
    <config-property-value>DEFAULT</config-property-value>
  </config-property>
```

3. Start the JBoss server using

   ```
   ${JBOSS_HOME}/bin/run.bat or ${JBOSS_HOME}/bin/run.sh
   ```

4. Deploy the JCA adapter rar file.

   You can copy the appropriate JVI JCA adapter rar file as below, in the deploy folder of your JBoss domain `$JBOSS_HOME/server/default/deploy.`

   `$VERSANT_ROOT/jvi/jca/vodLOCAL.rar,`

   `$VERSANT_ROOT/jvi/jca/vodNOTX.rar`,

   or `$VERSANT_ROOT/jvi/jca/vodXA.rar`

   Also copy the corresponding JBoss specific descriptors as below in the deploy folder of your JBoss domain `$JBOSS_HOME/server/default/deploy`.

   `$VERSANT_ROOT/jvi/jca/jboss/vodjvi_jca_local-ds.xml,`

   `$VERSANT_ROOT/jvi/jca/jboss/vodjvi_jca_notx-ds.xml,`

   or `$VERSANT_ROOT/jvi/jca/jboss/vodjvi_jca_xa-ds.xml`

BUILDING AND DEPLOYING THE SMP BEAN

Following are the steps to build and run the SMP Stateless and SMP Stateful examples using the JVI JCA Resource Adapter

WebLogic 9.x Configuration

1. Configure these common properties in `$VERSANT_ROOT/jvi/demo/jca/common.properties`

   ```
   DB_NAME=voddb
   ```

   For a 64 bit platform set OPTION64BIT to -d64.

   Create the database specified in common.properties before running the application:

   ```
   makedb voddb
   createdb voddb
   ```

2. Configure these properties in `$VERSANT_ROOT/jvi/demo/jca/weblogic.properties` as per your configuration.

   e.g.:

   ```
   JDK_HOME=E:/bea9.1/jdk150_04
   #WebLogic Home Directory
   WL_HOME=E:/bea9.1/weblogic91
   #WebLogic Domain Directory
   WL_USER_PROJECTS=E:/bea9.1/user_projects/domains
   #WebLogic domain name
   WL_DOMAIN=base_domain
   APP_DIR=${WL_USER_PROJECTS}/${WL_DOMAIN}/autodeploy
   ```

   You can now create a new domain in Weblogic 9.x using the Configuration wizard:

   Click Start -> Programs -> BEA Products -> Tools -> Configuration Wizard

3. To build and run the SMP stateless bean example:

   cd $VERSANT_ROOT/jvi/demo/jca/smp/stateless

   "`ant -f build_wl.xml deploy`" will compile, enhance all the sources and deploy the generated ear file.

   "`ant -f build_wl.xml run`" will run the client application.

4. To build and run the SMP stateful bean example:

cd $VERSANT_ROOT/jvi/demo/jca/smp/stateful

"`ant -f build_wl.xml deploy`" will compile, enhance all the sources and deploy the generated ear file.

"`ant -f build_wl.xml run`" will run the client application.

NOTE:- Instructions above assume that you are starting WebLogic server in development mode.

WebSphere 6.1 Configuration

To run the JVI JCA example using WebSphere 6.1

Make sure that the PATH is set for running `<WAS_HOME>/bin/ws_ant.(bat|sh)`.

1. Configure these common properties in `$VERSANT_ROOT/jvi/demo/jca/common.properties`

   ```
   DB_NAME=voddb
   ```

   For a 64 bit platform set OPTION64BIT to -d64.

   Create the database specified in common.properties before running the application

   ```
   makedb voddb
   createdb voddb
   ```

2. Modify these WebSphere properties in `$VERSANT_ROOT/jvi/demo/jca/websphere.properties`

   ```
   #WebSphere Home Directory
   WAS_HOME=C:/PROGRA~1/IBM/WebSphere/AppServer
   WAS_NODE=sysjavaw2kNode01
   WAS_CELL=sysjavaw2kNode01Cell
   ```

   #WebSphere server name in which the applications should be installed

   ```
   WAS_SERVER=server1
   ```

   #WebSphere virtual host name in which web applications should be installed

```
VIRTUAL_HOST=default_host
```

NOTE:- If "Enable Administrative Security" is checked at installation time then invocation of wsadmin script through `build_ws.xml` causes a window to display with a password prompt.

This can be avoided by following the steps given in this IBM WebSphere URL:

```
http://www-1.ibm.com/support/docview.wss?&uid=swg21142299
```

For a SOAP connection, edit the soap.client.props file:

```
$WAS_HOME/profiles/AppSrv01/properties/soap.client.props
```

Find the following two lines and add the appropriate user ID and password:

```
com.ibm.SOAP.loginUserid=
com.ibm.SOAP.loginPassword=
```

3. To build and run the SMP stateless bean example:

   ```
   cd $VERSANT_ROOT/jvi/demo/jca/smp/stateless
   ```

   "`ws_ant -f build_ws.xml deploy`" will compile, enhance all the sources and deploy the generated ear file.

   "`ws_ant -f build_ws.xml run`" will run the client application.

4. To build and run the SMP stateful bean example:
   cd $VERSANT_ROOT/jvi/demo/jca/smp/stateful

   "`ws_ant -f build_ws.xml deploy`" will compile, enhance all the sources and deploy the generated ear file.

   "`ws_ant -f build_ws.xml run`" will run the client application.

NOTE:- On Windows, if you observe the VException(`2994:SM_NOT_IN_USER_LIST`) error while running your application, please make sure that the Log On properties for the service "IBM WebSphere Application Server V6.1" reflect your account settings.

JBoss 4.0 Configuration

1. Configure these common properties in `$VERSANT_ROOT/jvi/demo/jca/common.properties`

   ```
   DB_NAME=voddb
   ```

   For a 64 bit platform set OPTION64BIT to -d64.

   Create the database specified in common.properties before running the application

   ```
   makedb voddb
   createdb voddb
   ```

2. Configure these properties in `$VERSANT_ROOT/jvi/demo/jca/jboss.properties` as per your configuration.

   e.g.

   ```
   #JBoss Home Directory
    JBOSS_HOME=E:/jboss-4.0.4.GA
    JBOSS_DEPLOY_DIR=${JBOSS_HOME}/server/default/deploy
   ```

3. To build and run the SMP stateless bean example:

   ```
   cd $VERSANT_ROOT/jvi/demo/jca/smp/stateless
   ```

   `"ant -f build_jboss.xml deploy"` will compile, enhance all the sources and deploy the generated ear file.

   `"ant -f build_jboss.xml run"` will run the client application.

4. To build and run the SMP stateful bean example:

   ```
   cd $VERSANT_ROOT/jvi/demo/jca/smp/stateful
   ```

   `"ant -f build_jboss.xml deploy"` will compile, enhance all the sources and deploy the generated ear file.

   `"ant -f build_jboss.xml run"` will run the client application.

Index