Java Versant Interface Tutorial

Your First Versant Application

VERSANT

Java Versant Interface Tutorial: Your First Versant Application

VERSANT

Table of Contents

Contact Information for Versant

You can obtain the latest information about Versant products by contacting either of our main office locations, visiting our web sites, or sending us email.

Versant Office Locations

Versant Corporation is headquartered in Redwood City, California. Versant GmbH is responsible for European operations and has headquarters in Hamburg, Germany.

Versant Corporation

255 Shoreline Drive, Suite 450
Redwood City, CA 94065
USA

1-800-VERSANT
+1.650.232.2400
+1.650.232.2401 (FAX)

Versant GmbH

Wiesenkamp 22b
D-22359 Hamburg
Germany

+49 (0)40 609 90 0
+49 (0)40 609 90 113 (FAX)

Versant Web Sites

For the latest corporate and product news, please visit the Versant web site at http://www.versant.com/.

The Versant Developer Center provides all the essential information and valuable resources needed for developers using Versant Object Database or Versant FastObjects including trial software and the Versant Developer forums. Visit the Versant Developer Center at http://developer.versant.com/.

Versant Email Addresses

For inquiries about about Versant products and services, send email to:

<mailto:info@versant.com>

For help in using Versant products, contact Technical Support at:

<mailto:support@versant.com> (Customer Services US)
or
<mailto:support@versant.net> (Customer Services Europe)

Please send feedback regarding this guide to:

<mailto:documentation@versant.com>

Chapter 1. Basic Procedures and Concepts

This tutorial contains a series of examples and explanations that demonstrate particular aspects of the transparent Java Versant Interface binding. We assume that you have installed the Versant Object Database 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 in your Java installation directory. Since this directory depends on the decisions you made at installation, the directory that contains the tutorial files is referred to as `TUT`. This is the directory `[VERSANT_ROOT]/demo/JVI/Tutorial` in your Versant installation.

This part of the tutorial shows you the basics. That is, how to make Java objects 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 persistence capable class

This step proceeds just as in regular Java development. No special Java code is necessary to make a class persistence capable.

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*. This means that nearly all of the work needed to allow objects to be persistently stored in a database 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.

Ensure that CLASSPATH is set properly before running the enhancer.

Run the application

As for 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.

1.1. Create a Database

All of the examples in this tutorial require a database in which to store objects. Thus, the first step is to create this database. The name of the database is arbitrary. In this tutorial we will use the name `tutdb`. You can choose another name if you wish 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.

Make the database directory

The first step in creating a new database is to create a subdirectory for it under the Versant root database directory. You do this with the Versant utility makedb. To create the database directory for the database `tutdb`, run the makedb utility with the database name as argument.

```
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 the *Versant Database Administration Manual*.

Create the database structure

Now you can create the database in the newly created database directory. To create the database, use the createdb utility.

```
createdb tutdb
```

This creates the storage structure and log files for the database system in the database directory `tutdb`.

1.2. Create a Persistence Capable 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 will be 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/src/tutorial/model/Person.java

```
package tutorial.model;

public class Person {
  public String name;
  public int age;

  public Person(String aName, int anAge) {
    name = aName;
    age  = anAge;
  }

  public String toString() {
    return "Person: " + name + " age: " + age;
  }
}
```

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.

1.3. 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 the `new` operator.
- Commit the transaction by ending the session.

TUT/src/tutorial/CreatePerson.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;


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 ();
  }
}
```

The following describes the application code.

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 JVI transparent binding, starting a database session is accomplished 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 passed 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 the *TransSession class* in the *Java Versant Interface Reference Manual*.)

Start a Transaction

A logical JDO transaction is started and changes made to objects in the context of the transaction are saved when the transaction is commited.

```
pm.currentTransaction().commit();
```

Starting a database session also starts a short transaction. All changes to persistent objects are written to the database only if the transaction is successfully commited.

Create a Persistent Object

Since the `Person` class will be designated as a persistence capable 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 short transaction to commit. To commit a transaction without ending the session, use the `TransSession.commit()` method.

1.4. 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 `CLASSPATH` environment variable 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`. The exact method for setting `CLASSPATH` depends on the system being used. Please refer to your Java documentation for correctly setting `CLASSPATH`.

An Ant script is provided for this tutorial. To compile the Java classes using Ant, use the following command-line calls.

```
cd TUT
ant clean
ant compile
```

1.5. Enhance the Java Classes

Enhancement is where the “magic” of the Java Versant Interface transparence 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 *post-processor* 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 by 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.

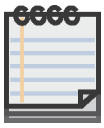
A configuration file, usually named `config.jvi`, controls the behavior of the enhancer. This file specifies the persistence category of each of the classes that are being enhanced. The configuration file for the example application above is as follows.

```
c tutorial.model.Person
a tutorial.CreatePerson
n**
```

category c. The letter `c` indicates that the class (in the example, the `tutorial.model.Person` class) is categorized as *persistence capable*. This means that objects of the class can be stored in the Versant database and the enhancer will modify the class so that persistence is possible.

category a. In the configuration file, the letter `a` indicates that a class is *persistence aware*. In the example, the `CreatePerson` class, is persistence aware. This means that `CreatePerson` instances will not be stored in the database but that the methods of class `CreatePerson` will be augmented to correctly work with other persistent objects, such as `Person` objects.

category n. The letter `n` indicates that the class should not be enhanced and `n**` means that the classes that have not been categorized with `a` or `c` will not be enhanced.



The configuration file supplied with the JVI installation has already been updated to work with this and the following examples. The configuration file is located in `[VERSANT_ROOT]/demo/JVI/Tutorial/src/config.jvi`.

The enhancer looks for `.class` files in an input directory and writes the 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/src/` and `TUT/build/` directories.

To enhance the Java classes, you can use the supplied Ant script.

```
cd TUT
ant enhance
```

Afterwards, the build directory will contain the enhanced `.class` files as well as additional files with names ending in `Pickler_Vj.class`. These classes assist in making the model objects persistent.

1.6. 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.

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 Versant 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.

```
ant runCreatePerson
```

The above command adds a single Person object to the tutdb database, with name Bob and age 28. This can be seen using the Versant db2tty utility.

```
db2tty -d tutdb -i tutorial.model.Person
```

For more information, on the db2tty utility refer to the *Versant Database Administration Manual*.

Chapter 2. 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.

2.1. 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/src/tutorial/CreatePersonWithRoot.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;


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 (taken from the command line).

To execute the CreatePersonWithRoot application run the supplied Ant script.

ant runCreatePersonWithRoot

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/src/tutorial/FindPersonWithRoot.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;
```

```
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 of `Person`. If no root has the given root name `findRoot()` throws an exception.

Use the Ant script to execute the FindPersonWithRoot application.

ant runFindPersonWithRoot

Running the application gives the following output.

```
Found Person: Mary age: 43
```

2.2. 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 the chapter *Fundamental JVI*, section *VQL Queries* in the *Java VERSANT Interface Usage Manual*.)

The following program finds all Person objects located in a database.

TUT/src/tutorial/FindPersonWithVQL.java

```
package tutorial;

import java.util.*;
import com.versant.trans.*;
import tutorial.model.*;


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 tutorial.model.Person");
    Enumeration e = query.execute();

    while (e.hasMoreElements()) {
      Person person = (Person) e.nextElement();
      System.out.println("Found " + person);
    }

    session.endSession();
  }
}
```

Finding all Person objects proceeds in three steps.

- First, the query object is constructed with a query that means “ find all objects from the Person class”.
- Next, the query is executed. This tells the Versant database to find the matching objects.
- 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.

Execute the FindPersonWithVQL application with Ant.

```
ant runFindPersonWithVQL
```

Running the application gives the following output.

```
Found Person: Bob age: 28
Found Person: Mary age: 43
```

Chapter 3. Changing Persistent Objects

This example shows how to modify 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/src/tutorial/IncreaseAge.java

```
package tutorial;

import java.util.*;
import com.versant.trans.*;
import tutorial.model.*;

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 tutorial.model.Person");
    Enumeration e = query.execute();

    while (e.hasMoreElements()) {
      Person person = (Person) e.nextElement();
      person.age++;
      System.out.println("Increasing " + person.name +
          "'s age to " + person.age);
```

```
    }

    session.endSession();
  }
}
```

To execute the IncreaseAge application, run **ant runIncreaseAge**.

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.

Chapter 4. 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.

The following points must be kept in mind when deleting objects from the database.

- 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 during transaction commit. 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/src/tutorial/DeletePersonWithRoot.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;

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 the database.

Execute the DeletePersonWithRoot application with **ant runDeletePersonWithRoot**.

Running the same program a second time generates an exception because the object was deleted from the database and the root name removed.

Chapter 5. 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 (logical object idetifier) 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 JVI.

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, Java 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 persistence application, we will use simple `Employee` and `Department` classes. The `Employee` class contains a reference to the `Department` class so that each employee belongs to one department. Similarly, the `Department` class contains a reference to the `Employee` class so that one employee manages each department.

Following are the class definitions.

TUT/src/tutorial/model/Employee.java

```
package tutorial.model;

public class Employee extends Person {

  public Department department;
  public double salary;

  public Employee(String aName, int anAge, double aSalary) {
    super(aName, anAge);
    salary = aSalary;
  }

  public String toString() {
    return "Employee: " + name + " age: " + age +
           " salary: " + salary + " " + department;
  }
}
```

TUT/src/tutorial/model/Department.java

```
package tutorial.model;

public class Department {
  String name;
  Employee manager;

  public Department(String aName, Employee aManager) {
    name    = aName;
    manager = aManager;
  }

  public String toString() {
    return "Department: " + name + " manager: " +
    ((manager != null) ?  manager.name : "nobody");
  }
}
```

The following application creates some `Employee` objects and stores them in the database.

TUT/src/tutorial/AddEmployees.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;


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();
  }
}
```

Execute the AddEmployees application with **ant runAddEmployees**.

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 tutorial.model.Employee
```

You can also use the FindPersonWithVQL application to find these `Employee` objects. Run **ant runFindPersonWithVQL** . The following is displayed.

```
Found Person: Bob age: 29
Found Employee: The Boss age: 42 salary: 11000.0 Department: Engineering manage
Found Employee: Jane Jones age: 24 salary: 80000.0 Department: Engineering mana
Found Employee: John Doe age: 25 salary: 75000.0 Department: Marketing manager:
```

```
Found Employee: Lois Line age: 36 salary: 70000.0 Department: Marketing manager: Lo
```

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. Run the IncreaseAge application with **ant runIncreaseAge**. The following is displayed.

```
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
```

Chapter 6. Transitive Persistence

So far you have seen just one way of storing objects persistently in the Versant database. That is, explicitly calling `makePersistent()` or `makeRoot()`. However, an object of a persistence capable class can also become persistent through a mechanism called *transitive persistence*. If a persistence capable object is referenced in a persistent object, it too becomes persistent.

To illustrate the concepts of persistence capable classes and transitive persistence we will use the following simple `LinkedList` class.

TUT/src/tutorial/LinkedList.java

```
package tutorial.model;

public class LinkedList {
  int label;
  LinkedList next_node;

  public LinkedList(int aLabel, LinkedList list) {
    label = aLabel;
    next_node = list;
  }

  public String toString() {
    return label + ((next_node == null) ? "" : " " + next_node);
  }
}
```

Now consider the following application which creates a linked list.

TUT/src/tutorial/CreateLinkedList.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;

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 labeled 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 none of the linked list nodes are made persistent.

To execute the CreateLinkedList application without a root name, run **ant runCreateLinkedList**.

By using the db2tty utility you can see that there are no `LinkedList` objects in the database.

```
db2tty -d tutdb -i tutorial.model.LinkedList
```

Now run the same application, this time with a root name, with the command **ant runCreateLinkedListWithRoot**.

Now the db2tty utility will show the five `LinkedList` nodes in the database. You can also see them with the following application.

TUT/src/tutorial/FindLinkedList.java

```
package tutorial;

import com.versant.trans.*;
import tutorial.model.*;

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 ();
  }
}
```

Execute the FindLinkedList application with **ant runFindLinkedList**. This will display the following.

```
Found List: 4 3 2 1 0
```

This means that even though only 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 demonstrates a very simple form of transitive persistence, where the linked objects were arranged in a regular, linear form, transitive persistence will apply to much more complicated linked structures as well. Any object of a persistence capable class becomes persistent if it can be reached from a persistent object by following links.

Chapter 7. 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* or SCO. 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 SCO. Each SCO 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 deserialized 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 separately. Second class objects, on the other hand, do not work easily with database queries (since the database has no knowledge of the Java serialization format) and involve some extra runtime serialization overhead. In addition, SCOs cannot be shared between first class objects because each SCO can have only one owner.

Fortunately, other than the restriction on sharing SCOs, you do not have to change the way in which you write your Java programs to determine if the tradeoffs are beneficial or harmful to the performance of your application. Only the configuration file need be changed.

As a simple example of using SCOs, 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 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
```

(The `d` stands for dependent, an alternate way of saying the object is second class.)

`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` (persistence capable), your application will behave in the same way, only the representation in the database will change.