Progress ObjectStore

ObjectStore PSE Pro

PSE Pro for Java User Guide

Release 7.1

PROGRESS SOFTWARE

PSE Pro for Java User Guide

PSE Pro for Java Release 7.1 for all platforms, August 2008

Contents

Preface

The *PSE Pro for Java User Guide* provides information and instructions for using PSE Pro for Java. A companion book, the *PSE Pro for Java Reference Guide* is available on line. It provides detailed information about the classes provided with PSE Pro. PSE Pro for Java allows you to write Java applications that create and access persistent data.

Audience

This book is for experienced Java programmers who want to write applications that use PSE Pro for Java.

Scope

This book supports Release 7.1 of PSE Pro for Java.

The Way This Book Is Organized

This book is organized as follows:

- Chapter 1, Introducing PSE Pro, on page 21, describes what PSE Pro does, shows the application architecture, and defines some important terms.
- Chapter 2, Example of Using PSE Pro, on page 31, describes the components your application must include to use PSE Pro.
- Chapter 3, Using Sessions to Manage Threads, on page 37, discusses how to initialize threads to use PSE Pro and how to use threads with PSE Pro sessions.
- Chapter 4, Managing Databases, on page 59, provides instructions for creating, opening, closing, and upgrading databases.
- Chapter 5, Working with Transactions, on page 79, describes how to start and end transactions.
- Chapter 6, Storing, Retrieving, and Updating Objects, on page 93, discusses the steps for storing, retrieving, and updating data.
- Chapter 7, Working with Collections, on page 139, provides information about PSE Pro utility collections. It also includes instructions for using the PSE Pro query facility.
- Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171, describes how to use the class file postprocessor to create persistence-capable classes.

- Chapter 9, Generating Persistence-Capable Classes Manually, on page 207, describes how to manually annotate classes you define so they are persistence capable.
- Chapter 10, Using the Java Dynamic Data (JDD) Classes, on page 225, describes how to use the JDD classes to model and store dynamic data without having to run the postprocessor or perform schema evolution.
- Chapter 11, Using Java Data Objects (JDO) with PSE Pro, on page 241 describes how to use the PSE Pro for Java implementation of the JDO specification. Storing persistent data with JDO is an alternative to storing persistent data in PSE Pro for Java databases.
- Chapter 12, Miscellaneous Information, on page 261, discusses serialization, `String` literals, building Java applets, and Java-supplied persistence-capable classes.
- Chapter 13, Tools Reference, on page 271, provides reference information for the PSE Pro utilities: `osjcfp`, `osjcheckdb`, `osjgcdb`, `osjshowdb`, and `osjversion`.
- Appendix, Packaging Your Application for End Users, on page 285, provides instructions for the files that you must include when you distribute your application.

Notation Conventions

This document uses the following notation conventions

Convention	*Meaning*
`Courier`	`Courier` font indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier`** font is used to emphasize particular code.
`Italic Courier`	*`Italic Courier`* font indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
Italic serif	In text, *italic serif* typeface indicates the first use of an important term.
[ ]	Brackets enclose optional arguments.
{ a | b | c }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
...	Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.

Progress Software on the World Wide Web

The Progress Software Web site (www.progress.com) provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

To obtain information about purchasing technical support, contact your local sales office listed at www.progress.com/about_us/worldwide_offices, or in North America call 1-781-280-4833. When you purchase technical support, the following services are available to you:

- You can send questions to ostore-support@progress.com. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at

  www.progress.com/support_main.
- You can file a report or question with Technical Support by going to www.progress.com/contact-customer-service.
- You can access the Technical Support Web site, which includes
  - A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
  - Solution Knowledge Base that you can browse and query.
  - Online documentation for all products.
  - White papers and short articles about using Progress products.
  - The latest versions of products, service packs, and publicly available patches that you can download.
  - Access to a support matrix that lists platform configurations supported by this release.
  - Support policies.
  - Local phone numbers and hours when support personnel can be reached.

Education Services

To learn about standard course offerings and custom workshops, use the Progress education services site (www.progress.com/objectstore/services).

To register for classes, call 1-800-477-6473 x4452. For information on current course offerings or pricing, send e-mail to classes@progress.com.

Searchable Documents

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the Progress Software Developers Network (PSDN) Web site at http://www.psdn.com/library/kbcategory.jspa?categoryID=1308. The site provides documentation for the most recent release and the previous supported release. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

Progress product development welcomes your comments about its documentation. Send any product feedback to ostore-support@progress.com. To expedite your documentation feedback, begin the subject with `Technical Support Issue; Doc:`. For example:

```
Subject: Technical Support Issue; Doc: Incorrect message on page 76
of reference manual
```

Third-Party Acknowledgments

This software makes use of the following third party products:

- Ant v1.6, Mortbay Jetty v6.1 and JXPath v 1.2. See the Apache License v2.0 in the installation directory in the docs/ThirdPartyLicenses folder for license information.
- Expat v9.5.1. Copyright © 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
- The Java SE Runtime Environment (JRE) Version 6, developed by and copyright Sun Microsystems. See the Sun Microsystems, Inc. Binary Code License for the Java SE Runtime Environment (JRE) Version 6 and THIRDPARTYLICENSEREADME.txt in the installation directory in the docs/ThirdPartyLicenses folder for license information.
- Jchart2d v2.2.0. The contents of these files are subject to the GNU Lesser General Public License v.2.1 (the ìlicenseî). You may not use these files except in compliance with the license. You may obtain a copy of the license in the installation directory in the docs/ThirdPartyLicenses folder and a copy of the license and source code of these files can be obtained through www.psdn.com by following the instructions set forth therein.
- JSON. Copyright © 2002 JSON.org. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above

copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The Software shall be used for Good, not Evil. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

- OpenDMK v 1.0-b2 and Restlet v 1.1m2. The contents of these files are subject to the Common Development and Distribution License (CDDL) Version 1.0 (the "License"). You may not use these files except in compliance with the License. You may obtain a copy of the License in the installation directory in the docs/ThirdPartyLicenses folder and a copy of the license and source code of these files can be obtained through www.psdn.com by following the instructions set forth therein.
- RSA Data Security, Inc. MD5 Copyright © 1991-2 RSA Data Security, Inc. Created 1991. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.
- Sun RPC v3.9 - Sun RPC is a product of Sun Microsystems, Inc. and is provided for unrestricted use provided that this legend is included on all tape media and as a part of the software program in whole or part. Users may copy or modify Sun RPC without charge, but are not authorized to license or distribute it to anyone else except as part of a product or program developed by the user. SUN RPC IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE. Sun RPC is provided with no support and without any obligation on the part of Sun Microsystems, Inc. to assist in its use, correction, modification or enhancement. SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY SUN RPC OR ANY PART THEREOF. In no event will Sun Microsystems, Inc. be liable for any lost revenue or profits or other special, indirect and consequential damages, even if Sun has been advised of the possibility of such damages. Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043.

- Tanuki Software Java Service Wrapper. See the Tanuki Software, Inc. Development Software License Agreement, Version 1.0 in the installation directory in the docs/ThirdPartyLicenses folder for license information. This product includes software and documentation components developed in part by Silver Egg Technology, Inc.("SET") prior to 2001. All SET components were released under the following license. Copyright © 2001 Silver Egg Technology. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub- license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
- Yahoo! User Interface Library - V 0.12.1. Copyright © 2006 Yahoo! Inc. All rights reserved. The name Yahoo! Inc. nor the names of its contributors may be used to endorse or promote products derived from this Software and products derived from this software without specific prior written permission of Yahoo! Inc. See the Yahoo! User Interface Library license V 0.12.1 in the installation directory in the docs/ThirdPartyLicenses folder for license information.

Chapter 1
Introducing PSE Pro

PSE Pro provides an application programming interface (API) that allows you to store Java objects persistently.

Contents

This chapter discusses the following topics:

What Is PSE Pro?

ObjectStore support for Java includes two products:

- PSE Pro
- Java interface to ObjectStore Development Client

PSE Pro is a personal storage editions for Java. Designed for single-user applications, it provides persistent storage for Java objects. Persistent data is available to programmers in such a way that it appears as familiar, normal Java objects. Persistent Java objects and regular Java objects are manipulated in the same way and behave in the same way.

PSE Pro is designed for applications that require persistent Java support for as much as 50 MB of data accessed by one user. The PSE Pro run-time library is written entirely in Java, uses less than 600 KB of disk space, and runs entirely within the application process. It supports transactions and provides access to objects without reading the entire database.

PSE Pro

- Supports hundreds of thousands of objects in a database and hundreds of megabytes of data in a database
- Provides better concurrency
- Allows multiple sessions (see page 24)

- Collects garbage in databases
- Provides collections interfaces and classes that are compatible with Java collection classes
- Supports queries and indexes
- Includes full database recovery from system failure
- Includes utilities for displaying information about the objects in a database and checking references in a database

The query facility is a separate software package. If you do not install the query facility, PSE Pro uses about 300 KB of disk space.

The Java interface to ObjectStore Development Client (referred to as ObjectStore) is for Java and C++ applications that require multiuser high-performance persistent storage for large databases with enterprise database features such as failover, on-line backup, fine-grained concurrency, and security.

ObjectStore can work well for distributed databases of virtually unlimited sizes and unlimited numbers of objects. The ObjectStore API is a superset of the PSE Pro APIs. In addition to providing all features provided by PSE Pro, ObjectStore supports the following:

- Applications that interface with databases and servers on local or remote machines
- Java applications and C++ applications that can access the same data
- Multiple concurrent sessions
- Multiple concurrent users
- Operating on multiple databases in a transaction
- Cross-database references
- On-line backup, failover, and archive logging

PSE Pro includes the `com.odi.odmg` package, which provides an Object Data Management Group (ODMG) binding. This binding includes classes for `Database` and `Transaction` that closely follow the ODMG specification. The package also includes the `com.odi.odmg.Collection` interface, persistence-capable classes that implement the `Collection` interface, and ODMG exception classes. See the `com.odi.odmg` package in the *Java API Reference*.

What PSE Pro Does

PSE Pro provides an API that allows a program to

- Start and end sessions to allow threads to use the PSE Pro API
- Create, open, close, and destroy databases
- Start, commit, and abort transactions to access data in the database
- Read and write database roots, which provide starting points for navigating to persistent objects

- Store objects in a database and retrieve and update those objects

PSE Pro can recover from an application failure or system crash. If a failure prevents some of the changes in a transaction from being saved to disk, PSE Pro ensures that none of that transaction's changes are saved in the database. When you restart the application, the database is consistent with the way it was before the transaction started.

PSE Pro does not have facilities for protection against media failure.

Benefits of Using PSE Pro

PSE Pro provides a convenient and complete API for storing and sharing Java objects among users, hosts, and programs. After you define persistence-capable classes (classes whose instances can be stored in a database), writing a PSE Pro application is like writing any other Java application.

PSE Pro allows you to quickly read or modify portions of your persistent data. You are not required to read in all persistent data when you just want to look at a subset. This reduces start-up and transaction commit times and allows you to run much larger Java applications without increasing the amount of memory or swap space on the system.

When you access persistent data inside a transaction, PSE Pro ensures that your results are not compromised by other users sharing the data. If something goes wrong, or if you determine that you do not want to keep changes, you can abort the transaction. In that case, PSE Pro restores the database to the state it was in before the transaction started. This makes recovering from exceptions or failures straightforward.

Description of PSE Pro Architecture

PSE Pro is a Java library that runs entirely within your Java virtual machine process. A PSE Pro database consists of three files. PSE Pro uses the standard Java class `java.io.RandomAccessFile` to access a database. There is no client/server distinction.

There is no PSE Pro server. PSE Pro is just accessing a file on the file system. PSE Pro has no special privileges. If PSE Pro can read or write a file, so can any other user using ordinary file I/O.

PSE Pro does not maintain a cache. Instead, PSE Pro uses weak references to objects in memory. If your application does not maintain a reference to an object in memory, the Java garbage collector is free to collect that object. PSE Pro does provide `retain` options that allow restricted use of objects outside transactions. See Committing Transactions to Save Modifications on page 111.

Definitions of PSE Pro Terms

This section describes the following terms, which you must be familiar with to use PSE Pro:

- Session
- Persistence Capable
- Persistent Object
  - Hollow persistent objects
  - Active persistent objects
  - Stale persistent objects
- Persistence Aware
- Transient Object
- Transitive Persistence
- Annotations
- Database Roots

Session

A *session* allows the use of the PSE Pro API. PSE Pro uses the abstract `com.odi.Session` class to represent sessions.

Your application must create a session before it can use any of the PSE Pro API. After a session is created, it is an active session. A session remains active until your application or PSE Pro terminates it. After a session is terminated, it is never used again. You can, however, create a new session.

A session creates a context in which you can create a transaction, access a database, and manipulate persistent objects. A session consists of a set of persistent objects and a set of PSE Pro API objects such as a `Database` and a `Transaction`. In a single Java VM process, PSE Pro and ObjectStore allow multiple concurrent sessions.

Separate Java virtual machines can each run their own sessions at the same time. In addition, if you are using PSE Pro or ObjectStore, separate Java virtual machines can each run multiple sessions at the same time. See How Sessions Keep Threads Organized on page 37.

Persistence Capable

The term *persistence capable* refers to the capacity of an object to be stored in a database. If you can store the instances of a class in a database, the class is a persistence-capable class and the instances are persistence-capable objects.

The definition of a persistence-capable class includes specific annotations required by PSE Pro. After you compile class definitions, you run the PSE Pro class file postprocessor on the compiled classes to add the annotations that make the classes persistence capable. For more information, see Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171. In unusual circumstances, you might

choose to add the annotations to the Java source file manually. For more information, see Chapter 9, Generating Persistence-Capable Classes Manually, on page 207.

You must explicitly postprocess or manually annotate each class that you want to be persistence capable. The capacity for an object to be stored in a database is not inherited when you subclass a persistence-capable class.

Some Java-supplied classes are persistence capable. Other classes are not and cannot be made persistence capable. A third category of classes can be made persistence capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 261.

Persistent Object

A *persistent object* is a representation of an object that is stored in a database. After an application retrieves an object from the database, the application works with the persistent object in the Java environment.

A persistent object always exists in one of three states:

- Hollow
- Active
- Stale

Methods you call can change the state of a persistent object.

Hollow persistent objects

A *hollow persistent object* has the same structure as the object in the database that it represents. A hollow object contains the same fields as the object in the database that the persistent object represents, but the fields of the hollow object usually do not contain values corresponding to those values stored in the database. The values for the fields in a hollow object might be `null` or the values from a previous transaction.

When your application acquires a reference to an object that has not yet been read in from the database, PSE Pro generates a hollow object as a placeholder for that object. PSE Pro does not actually read in the contents of the object until your application tries to read, write, or invoke a method on the object.

When your application accesses a hollow object, PSE Pro turns it into an active persistent object. PSE Pro retrieves the contents of the object from the database and stores them in the fields of the hollow object, which makes it an active persistent object. This process is referred to as initialization. In most applications, this happens automatically, because the postprocessor inserts the required calls.

Obtaining an object from a database root always results in a hollow object. If you get the same root three times, the object it identifies is still hollow. You must access the object to make it active.

After an application accesses the contents of persistent objects, the objects that the persistent object references are hollow objects unless their contents were accessed previously. For example, suppose that you have the following class:

```
class A {
  B b;
}
```

When you obtain a reference to an instance of `A`, PSE Pro creates a hollow `A` object to represent that instance. When you read or update the instance of `A`, PSE Pro turns it into an active object and creates a hollow `B` object to represent the referred-to instance of `B`. If you then read or update the instance of `B`, PSE Pro makes that hollow object into an active object and creates hollow objects for any objects referred to by `B`.

Active persistent objects

A persistent object must be active before an application can read or update it. An *active persistent object* starts as an exact copy of the object that it represents in the database. The contents of an active object are available to be read by the application and might be available to be modified. If an active object is updated by the application, it is no longer identical to the object in the database that it represents.

When a persistent object is active, PSE Pro internally flags it as either clean or dirty. An active object is marked initially as clean when its contents are read into memory. At this point, PSE Pro recognizes that the contents of the persistent object match the contents of the object in the database. An active object is dirty when it is a modified version of the stored object that the active object represents. When you modify an object, PSE Pro automatically changes the flag from clean to dirty. The class file postprocessor inserts the code that makes a persistent object clean or dirty.

For example, suppose that you have an instance of a `Person` object in which the `age` field has the value `30`. When you read this object, it is in the clean state. If you modify the value of `age`, even if the new value you assign is `30`, the object is then in the dirty state.

Stale persistent objects

A *stale persistent object* is no longer valid. Its fields have default values or values left over from previous transactions and should not be used. An object becomes stale when an application calls

- `Transaction.commit()` on the transaction in which the object could be read or modified, and the call
  - Does not specify a `retain` argument, or it specifies `ObjectStore.RETAIN_STALE`
  - And `Transaction.setDefaultCommitRetain()` has not been set or it specifies `ObjectStore.RETAIN_STALE`
  - And `Transaction.setDefaultRetain()` has not been set or it specifies `ObjectStore.RETAIN_STALE`

  (There is not an API that sets a default `retain` value for the `evict()` or the `destroy()` methods. You can only set a default `retain` value for the `abort()`, `checkpoint()`, and `commit()` methods.)
- `Transaction.abort()` on the transaction in which the persistent object could be read or modified, and the call
  - Does not specify a `retain` argument or it specifies `ObjectStore.RETAIN_STALE`
  - And `Transaction.setDefaultAbortRetain()` has not been set or it specifies `ObjectStore.RETAIN_STALE`
  - And `Transaction.setDefaultRetain()` has not been set or it specifies `ObjectStore.RETAIN_STALE`

- `Transaction.checkpoint()` on the transaction in which the persistent object could be read or modified, and the call
  - Does not specify a `retain` argument or it specifies `ObjectStore.RETAIN_STALE`
  - And `Transaction.setDefaultCommitRetain()` has not been set or it specifies `ObjectStore.RETAIN_STALE`
  - And `Transaction.setDefaultRetain()` has not been set or it specifies `ObjectStore.RETAIN_STALE`
- `ObjectStore.evict()` on the object and the call does not specify a `retain` argument, or it specifies `ObjectStore.RETAIN_STALE`
- `ObjectStore.destroy()` on the object

If an application tries to read or update a stale object, PSE Pro signals `ObjectException`. An application must not invoke any instance method on a stale object.

Persistence Aware

If the methods of a class can operate on fields of persistent objects, but instances of the class itself are not persistence capable, the class is *persistence aware*.

Typically, if you want a class to be persistence aware, you run the postprocessor on it to put in the required annotations. See Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171. Occasionally, you might choose to annotate the class manually to make it persistence aware. See Chapter 9, Generating Persistence-Capable Classes Manually, on page 207.

When a method accesses fields in a persistent object, PSE Pro checks to ensure that the data has been read from the database. This checking is done by calls that the postprocessor inserts in your code. These are the annotations mentioned in the previous paragraph. The annotations are calls to the `ObjectStore.fetch()` or `ObjectStore.dirty()` method.

Every persistence-capable and persistence-aware class must have these annotations. Persistence-capable classes also include many other annotations.

A class must be persistence aware only if it directly accesses the fields of a persistence-capable object. This includes access of elements of a persistent array. If your persistence-capable classes have only private fields and do not return arrays that might be persistent, other classes can call methods on the persistence-capable object without being persistence aware.

Transient Object

A *transient object* is an object that is not already in a database.

Transitive Persistence

When an application commits a transaction, PSE Pro stores in the database any transient objects that can be reached transitively from any persistent object. This is the process of *transitive persistence*. Transient objects that are referenced by persistent objects become persistent when the transaction commits. For this to work, the transient objects must be persistence capable.

Annotations

The class file postprocessor annotates classes you define so that they are persistence capable. This means that the postprocessor makes a copy of your class files, overwrites your original class files or places them in a directory that you specify, and adds byte code instructions (*annotations*) that are required for persistence. Complete information about annotations is in Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171.

Occasionally, you might want to annotate your code manually. Information you need to do this is in Chapter 9, Generating Persistence-Capable Classes Manually, on page 207.

Database Roots

A *database root* provides a way to associate a name with an object in a database. Applications use database roots to locate one or more persistent objects for performing queries or navigating to other persistent objects. When you make an object the value of a persistent database root, doing so establishes the object as persistent and makes the objects it refers to available for transitive persistence.

At any given time, a database root is either associated with one database or it is null. You can change the database with which a root is associated. Information about database roots is in Working with Database Roots on page 96.

Prerequisites for Using PSE Pro

To use PSE Pro, you must

- Be an experienced programmer familiar with the Java language.
- Have a supported platform as defined in the support matrix on the Technical Support web site (`www.objectstore.net/support/matrix`). This Web page contains an up-to-date list of all supported and maintained platforms. Refer to the support matrix if you are in any doubt whether your compiler or operating system are supported.
- Have available a supported Java platform. The compiler must conform to JavaSoft specifications. The Java VM must be among those supported by PSE Pro. You cannot use a supported compiler with an unsupported VM.
- Have a valid license file.

  To run PSE Pro for Java, the PSE Pro for Java runtime must be able to access a valid PSE Pro for Java license file. The name of this file must be `license.txt`. During installation, the installation script copies the license file you specify to the `etc/license.txt` file in the PSE Pro for Java installation directory. At runtime, PSE Pro for Java looks for a license file as follows:

  a If you set the `com.odi.OStoreLicenseFile` Java property, then the installation script uses that property value as the absolute path to the PSE Pro for Java license file.

  b If you do not set the `com.odi.OStoreLicenseFile` property, the installation script determines the location of the `pro.jar` file (`pro_g.jar` for debug builds) and looks for the license file in the `../etc` directory.

  If the license file is missing, invalid, or expired, then a fatal license exception occurs and the installation script terminates.

Chapter 2
Example of Using PSE Pro

This chapter provides a simple example of a complete PSE Pro program. The code for this example is in the `com\odi\demo\people` directory provided with PSE Pro.

Contents

This chapter discusses the following topics:

Overview of Required Components

The sample program stores information about a few people, then retrieves some of the information from the database and displays it. The program shows the components you must include in your application so that it can use PSE Pro. These components are

- Create a session.The example calls the `Session.create()` method to start a nonglobal session. See page 41.
- Join a thread to a session. The example calls the `Session.join()` method to associate this thread with the session. See page 46.
- Create or open a database. The example creates the `person.odb` database and uses the `db` variable to refer to it. See page 59.
- Start and commit transactions as needed. The example uses one transaction to store the objects in the database. It then uses a second transaction to retrieve the stored objects. See page 79.
- Create a database root, which provides a starting point for accessing objects in the database. The example creates a root with the name `"Tim"` and associates it with the `tim` instance of the `Person` class. See page 96.
- Store objects referenced by a root in the database. The example stores `sophie` and `joseph` in the database when the transaction is committed. See page 94.
- Use a database root to retrieve objects from a database and do something with them. The example starts a new transaction, retrieves `tim`, and displays a line of information. See Retrieving Persistent Objects on page 95.
- End the session. The example calls the `Session.terminate()` method. This closes the open database and shuts down PSE Pro. See page 43.

When you write a PSE Pro program, you write it as though classes are persistence capable. However, a program cannot store objects persistently until you run the PSE Pro-provided class file postprocessor. The postprocessor generates annotated versions of the class files. The annotated version of the class definition is persistence capable. You run the postprocessor after you compile the program and before you run the program.

Sample Code

```
package com.odi.demo.people;
// Import the com.odi package, which contains the API:
import com.odi.*;
public
class Person {

  // Fields in the Person class:

  String name;
  int age;
  Person children[];

  // Main:
  public static void main(String argv[]) {

    try {
      String dbName = argv[0];

      // The following line starts a nonglobal session and
      // joins this thread to the new session. This allows the
      // thread to use PSE Pro.
      Session.create(null, null).join();

      Database db = createDatabase(dbName);
      readDatabase(db);
      db.close();
    }

  // The following shuts down PSE Pro.
    finally {
    Session.getCurrent().terminate();
  }
  }

  static Database createDatabase(String dbName) {

    // Attempt to open and destroy the database specified on the
    // command line. This ensures that the program creates a
    // new database each time the application is called.

    try {
  Database.open(dbName, ObjectStore.UPDATE).destroy();
    }  catch (DatabaseNotFoundException e) {
    }

    // Call the Database.create() method to create a new database.

  Database db = Database.create(dbName,
  ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
    // Start an update transaction:

    Transaction tr = Transaction.begin(ObjectStore.UPDATE);
    // Create instances of Person:

    Person sophie = new Person("Sophie", 5, null);
    Person joseph = new Person("Joseph", 1, null);
```

```
  Person children[] = {sophie, joseph};
  Person tim = new Person("Tim", 35, children);

  // Create a database root and associate it with
  // tim, which is a persistence-capable object.
  // PSE Pro uses a database root as an entry
  // point into a database.

  db.createRoot("Tim", tim);

  // End the transaction. This stores the three person
  // objects, along with the String objects representing
  // their names, and the array of children, in the database.
  tr.commit();

  return db;

}

static void readDatabase(Database db) {

  // Start a read-only transaction:

  Transaction tr = Transaction.begin(ObjectStore.READONLY);

  // Use the "Tim" database root to access objects in the
  // database. Because tim references sophie and joseph,
  // obtaining the "Tim" database root allows the program
  // also to reach sophie and joseph.
  Person tim = (Person)db.getRoot("Tim");
  Person children[] = tim.getChildren();
  System.out.print("Tim is " + tim.getAge() + " and has " +
    children.length + " children named: ");
  for (int i=0; i < children.length; i++) {
    String name = children[i].getName();
    System.out.print(name + " ");
  }
  System.out.println("");
  // End the read-only transaction.
  // This form of the commit method ends the accessibility
  // of the persistent objects and makes the objects stale.
  tr.commit();
}

// Constructor:

public Person(String name, int age, Person children[]) {
   this.name = name; this.age = age; this.children = children;
}

public String getName() {return name;}
public void setName(String name) {this.name = name;}
public int getAge() {return age;}
public void setAge(int age) {this.age = age;}
public Person[] getChildren() {return children;}
public void setChildren(Person children[]) {
    this.children = children;
}

// This class is never used as a persistent hash key, so
```

```
  // include the following definition. If you do not, then
  // when you run the postprocessor it is unclear whether or
  // not you intend to use the class as a hash code.
  // Consequently, the postprocessor inserts a hashCode
  // function for you. The following definition avoids this.
  public int hashCode() {
    return super.hashCode();
  }

}
```

Before You Run the Program

Before you can run the sample program, you must

- Add an entry to your `CLASSPATH` environment variable
- Compile the source file
- Run the postprocessor on the `.class` file

Adding an Entry to CLASSPATH

In your `CLASSPATH` environment variable, you already have two entries related to PSE Pro:

- One entry for the `pro.jar` file to use PSE Pro
- One entry for the `tools.jar` file to use the class file postprocessor and other database tools

Ensure that these `.jar` files are explicitly in your class path. An entry for the directory that contains them is not sufficient.

Another entry is required for you to be able to build and run the program. This entry names the PSE Pro installation directory and allows PSE Pro to locate the annotated class files when you run the program.

For example, on Windows, if you place the PSE Pro distribution in the `c:\ODI\PSEProJ` default directory, you need the following entries:

```
c:\ODI\PSEProJ\lib\pro.jar;
  c:\ODI\PSEProJ\lib\tools.jar;
  c:\ODI\PSEProJ
```

On UNIX, if you place the PSE Pro distribution in `/usr/local/odi/pseproj` (the default), you need:

```
/usr/local/odi/pseproj/lib/pro.jar:
  /usr/local/odi/pseproj/lib/tools.jar:
  /usr/local/odi/pseproj
```

Compiling the Program

To compile the program, change to the `com\odi\demo\people` directory and enter

```
javac *.java
```

As output, the `javac` compiler produces the byte code class file `Person.class`.

Running the Postprocessor

You must run the class file postprocessor to make the `Person` class persistence capable. The postprocessor generates new annotated class files. After you run the postprocessor, your program uses the annotated class files rather than the original class files.

Ensure that the `bin` directory that contains the `osjcfp` executable is in your path, as noted in the `README` file in the installation directory and the postprocessor documentation. See Preparing to Run the Postprocessor on page 175.

To run the postprocessor, enter

```
osjcfp -dest . -inplace Person.class
```

The `-dest` option specifies a destination directory for the annotated files. It is a required option. The `-inplace` option specifies that the postprocessor should overwrite the original class files. When you specify the `-inplace` option, the postprocessor ignores the `-dest` option.

The result from the `osjcfp` command shown above is the annotated class file `com\odi\demo\people\Person.class`.

The `-inplace` option is the best choice for this example. However, when you are in an iterative development cycle, it is best not to specify `-inplace`. During development, putting the postprocessed files in a different directory avoids errors.

Running the Program

Run the program as a Java application. Following is a typical command line:

```
java com.odi.demo.people.Person person.odb
```

The argument is the pathname of the database's `.odb` file, namely `person.odb`. The application also creates `person.odt`, and `person.odf`, and these files form the database. You can specify any pathname you want, as long as the file name ends with `.odb`. This example uses a relative pathname, so PSE Pro creates the files in the current working directory.

The expected output is

```
Tim is 35 and has 2 children named: Sophie Joseph
```

Also, the example application creates or replaces the `person.odb` database in the current directory.

Chapter 3
Using Sessions to Manage Threads

This chapter provides information about how to manage the threads in your application. Sample code that uses threads is in `com\odi\demo\threads`.

Contents

This chapter discusses the following topics:

How Sessions Keep Threads Organized

For a thread to use PSE Pro, it must be associated with a session. To use threads with PSE Pro, you must create at least one session and you must understand how to work with sessions.

If you try to use the PSE Pro API and you have not created a session, PSE Pro signals `NoSessionException`.

This section discusses the following:

- What Is a Session?
- How Are Threads Related to Sessions?
- What Is the Benefit of a Session?
- What Kinds of Sessions Are There?

What Is a Session?

A session allows the use of the PSE Pro API. PSE Pro uses the abstract `com.odi.Session` class to represent sessions.

Your application must create a session before it can use any of the PSE Pro API. After a session is created, it is an active session. A session remains active until your application or PSE Pro terminates it. After a session is terminated, it is never used again. You can, however, create a new session.

A session creates a context in which you can create a transaction, access a database, and manipulate persistent objects.

Concurrent sessions

In a single Java VM, PSE Pro and ObjectStore allow multiple concurrent sessions.

If you are using ObjectStore or PSE Pro, separate Java virtual machines can each run multiple sessions at the same time.

The default behavior for PSE Pro is that at any one time only one Java VM process can access a database. See Description of Concurrency Rules on page 87.

How Are Threads Related to Sessions?

At any given time, an active session has zero or more associated *threads*. Any number of threads can join a session. Each thread can belong to only one session at a time.

At any given time, each thread is either joined to a single session or not joined (not associated) to a session. A thread that is not associated with a session can join a session. A thread that is associated with a session can leave the session to end its association with that session. It can rejoin the session at a later time or it can join another session.

For a thread to use the PSE Pro API, it must automatically or explicitly be associated with a session. All threads that join the same session cooperate with each other. PSE Pro does not prevent cooperating threads from accessing the same object. Consequently, it is your responsibility to identify code segments that must be synchronized. To successfully call the `Session.join()` method to join a session, a thread must not already be associated with any session.

The *current thread* is the thread that you are making a call from. The *current session* is the session that the current thread belongs to.

What Is the Benefit of a Session?

The benefit of a session is apparent when you want to have more than one session. Two sessions in the same Java process allow you to perform two distinct activities that involve PSE Pro. Each session has a clean, isolated view of the database. If you want to have two or more independent transactions going on at the same time, you can use two or more sessions. Concurrent sessions can be accessing the same database or different databases.

When two sessions are accessing the same object in the database, there are two distinct persistent objects. Each session has its own persistent object, which is a copy

of the object in the database. At least initially, these two persistent objects have the same content.

Each session has its own set of persistent objects and API objects. In most circumstances, the threads of session A are not allowed to operate on the persistent objects of session B. An exception to this rule is described in Multiple Representations of the Same Object on page 50.

Independent threads

A need for many different independent transactions normally arises because you have many Java threads with different things occurring in each one. Typically, this happens with a multithreaded application server, in which there are many threads. Each thread serves a different client, so you might want to have many threads. Each thread runs a separate transaction, and each thread is separate from each other thread.

Cooperating threads

On the other hand, there are times when you have multiple threads that are cooperating on some database task and must operate on the same objects at the same time. In this case, you might want two different Java threads to participate in the same transaction.

Controlling the threads that cooperate

Sessions allow you to control the threads that cooperate in a transaction and work in independent transactions. A session groups together a set of cooperating threads. Each session has a sequence (in time) of transactions and a set of associated threads that participate in these transactions.

There is a many-to-one relationship between threads and sessions. That is, any number of threads can belong to one session.

Example of cooperating threads

A common case of cooperating threads arises when you are writing a Java applet. In an applet, there are calls to different parts of your program in different threads. You have to specify for PSE Pro that all these threads are part of the same session. This allows them to operate on the same objects and in the same transactions. A similar situation exists when you use RMI and CORBA servers; that is, there is a control mechanism that calls your methods in different threads.

What Kinds of Sessions Are There?

An active session can be a global session or a nonglobal session. PSE Pro provides two kinds of sessions because when you need only one session, PSE Pro can do many things for you automatically.

Joining threads to sessions

As mentioned earlier, before you can use PSE Pro, you must create a session. For a thread to use PSE Pro, it must join a session. In a global session, an unassociated thread that makes a call to the PSE Pro API joins the session automatically. In a nonglobal session, this happens only when the call implies the session. See Rules for Joining a Thread to a Session Automatically on page 45. Otherwise, you must explicitly add the thread to a session.

Number of sessions

An active global session is the only session in the Java VM. With PSE Pro and ObjectStore, you can have multiple nonglobal sessions or one global session in a Java VM.

Global sessions

Global sessions make programming easier because you need not know the PSE Pro APIs for associating threads with sessions. All threads that make PSE Pro API calls join the one global session automatically.

The drawback is that you can have only one session. If you change your program in the future to use multiple sessions, you might have to go back and put in API calls to associate threads with the appropriate session. If you think you might use multiple sessions in the future, it would probably be a good idea to prepare for that by using a nonglobal session and explicitly joining the session in each thread.

Creating Sessions

When you create a session, you initialize PSE Pro for use by the threads that become associated with that session. You can create a session in the following ways:

- Call the Session.createGlobal() method to create a global session.
- Call the Session.create() method to create a nonglobal session.

Regardless of how you create a session, you can specify a number of PSE Pro properties when you create the session. These properties determine how PSE Pro behaves in a variety of situations.

Creating Global Sessions

When the session is a global one and a thread that is not associated with the session calls a PSE Pro API, PSE Pro automatically joins the thread to the session. After you create a global session, you need not be concerned about joining threads to the session.

To create a global session, call the `Session.createGlobal()` method. The method signature is

```
public static Session createGlobal(String host,
  java.util.Properties properties)
```

This method creates and returns a new session and designates the session as a global session. No threads are joined to this session yet. Any thread, including the thread that creates the session, automatically joins the session the first time the thread uses PSE Pro.

PSE Pro ignores the first parameter; you can specify null. The second parameter specifies null or a property list. See Description of PSE Pro Properties on page 52.

If you try to create a global session when there is already an active session, PSE Pro signals `ObjectStoreException`.

To obtain a global session, call `Session.getGlobal()`. The method signature is

```
public static Session getGlobal()
```

If the global session is active, PSE Pro returns it. Otherwise, PSE Pro returns null.

Creating Nonglobal Sessions

You can create a nonglobal session. The difference between a global and nonglobal session is that in a nonglobal session

- PSE Pro does not join all unassociated threads to the session automatically.
- Multiple nonglobal sessions can exist in the same Java VM for ObjectStore and PSE Pro.

Joining threads to sessions

For PSE Pro to join an unassociated thread to a nonglobal session automatically, the thread must be making a PSE Pro API call that implies a session. See Rules for Joining a Thread to a Session Automatically on page 45.

You must explicitly join a thread to a session before that thread can call a PSE Pro API that does not imply a session. See Explicitly Associating Threads with a Session on page 46.

Method signature

The method signature for creating a nonglobal session is

```
public static Session create(String host,
  java.util.Properties properties)
```

This method creates and returns a new session. PSE Pro ignores the `host` argument; specify null. The second argument specifies null or a property list. See Description of PSE Pro Properties on page 52.

PSE Pro does not join the calling thread to the session. If you are using ObjectStore or PSE Pro, a thread can belong to a nonglobal session and call a method that creates another nonglobal session.

Session name

PSE Pro generates a name for the session and never reuses that name for the lifetime of the process in which the session was created. If you want to specify a particular name, use the following overloading to specify a unique session name:

```
public static Session create(String host,
  java.util.Properties properties,
  String name)
```

PSE Pro uses the session name in debugging messages. The `Session.getName()` method returns the name of the session.

Exception conditions

If you call `Session.create()` when there is an active global session, PSE Pro signals `ObjectStoreException`.

Working with Sessions

After you create a session, you need to know how the session functions with regard to transactions. You also need to know about the operations you can perform on the session. This section discusses

- Sessions and Transactions
- Shutting Down Sessions
- Obtaining a Session
- Determining Whether a Session Is Active

Sessions and Transactions

At any given time, a session has one associated transaction in progress or it does not have any associated transaction. Each transaction is associated with exactly one active session.

When a session is created, there is no associated transaction. While a session is active, an application can start, then commit or abort, one transaction at a time per session. Over time, a session is associated with a sequence of transactions.

If a transaction is in progress when an application or PSE Pro shuts down the session, PSE Pro aborts the transaction as part of the shutdown process.

Within a session, one database at a time can be open. If you need to have multiple databases open at the same time, you must have a session for each database that needs to be open.

In PSE Pro, multiple sessions within a Java VM can have simultaneous independent transactions. These transactions can be doing any one of the following:

- Each transaction can access a different database. In this case, all sessions can have read-only transactions, or all sessions can have update transactions, or there can be any combination of read-only and update transactions.
- Some transactions can access the same database, while other transactions access a different database or several different databases. In this case, any number of sessions can have read-only transactions against the same database at the same time. If at least one session has a read-only transaction against a database, no session can have an update transaction against that database.

Only one session at a time can have an update transaction against a database. When a session has an update transaction against a database, no other session can have a read-only transaction against that database.

See also Description of Concurrency Rules on page 87 and Preventing More Than One Process from Accessing a Database on page 90.

Transaction in progress?

To determine whether there is a transaction in progress, call the `Session.inTransaction()` method. The method signature is

```
public boolean inTransaction()
```

If there is a transaction associated with the session, this method returns `true`. If there is no transaction associated with the session, this method returns `false`. If the session has been terminated, PSE Pro signals `NoSessionException`.

Obtaining the associated transaction

To obtain the transaction associated with a session, call the `Session.currentTransaction()` method. The method signature is

```
public Transaction currentTransaction()
```

If the session has been terminated, PSE Pro signals `NoSessionException`. If no transaction is associated with the session, PSE Pro signals `NoTransactionInProgressException`.

Obtaining the transaction's session

To obtain the session associated with a transaction, call the `Transaction.getSession()` method. The method signature is

```
public Session getSession()
```

Shutting Down Sessions

When your application opens a database, PSE Pro creates a directory that functions as a lock against other Java VM processes. The name of the directory is *`database_name`*`.odx`

PSE Pro maintains this directory in the same directory as your database. When your application closes the database or terminates the session (this closes the database) that opened the database, PSE Pro deletes the `.odx` directory, which releases the lock. Consequently, you must close a database to make that database available to other processes.

To do this, ObjectStore recommends that you call the `Database.close()` or `Session.terminate()` method in the `finally` clause of your program. If you do not close a database, the database remains locked and other VM processes cannot access it. For an example, see Preventing More Than One Process from Accessing a Database on page 90.

Another reason to shut down a session is to release the Java objects associated with the session.To shut down a session, call the `Session.terminate()` method. The method signature is

```
public void terminate()
```

It does not matter whether the session is a global session or a nonglobal session; PSE Pro shuts down the session. If there are no other sessions, no thread can use the PSE Pro API until there is a new active session. The terminated session is never reused.

Transaction in progress

If the session you shut down has an associated transaction, PSE Pro aborts the transaction. If the session has already been terminated, PSE Pro does nothing. If the session has any associated threads, PSE Pro causes them to leave the session. If the session has an open database, PSE Pro closes it.

If PSE Pro signals `FatalException`, this shuts down the session.

Obtaining a Session

You can obtain a session with a call to any of the methods in the following list:

- `Placement.getSession()`
- `Session.getCurrent()`
- `Session.getGlobal()`
- `Session.of(object)`
- `Session.ofThread(thread)`
- `Transaction.getSession(thread)`

Determining Whether a Session Is Active

To determine whether a session is active, call the `Session.isActive()` method. The method signature is

```
public boolean isActive()
```

If the session is active, this method returns `true`. If the session has been terminated, this method returns `false`.

Associating Threads with Sessions

To help you associate threads with sessions, this section discusses

- Joining Threads to a Session Automatically
- Associating a Persistent Object with a Session
- Rules for Joining a Thread to a Session Automatically
- Examples of Calls That Imply Sessions
- Examples of Calls That Do Not Imply Sessions
- Explicitly Associating Threads with a Session

There is a bug in the software that prevents threads from being joined to sessions automatically. As a work around, you must explicitly join each thread to a session. See the *ObjectStore Release Notes* for details. This bug will be fixed in a future release.

Joining Threads to a Session Automatically

Whether a thread can join a session automatically depends on

- Whether the session is global or nonglobal
- Whether the API call that the thread is making implies a session

Global sessions

When there is a global session, an unassociated thread that makes a call to the PSE Pro API joins the global session automatically, if necessary. In the following situations, it might not be necessary to join the thread to the session:

- An unassociated thread calls a method on a transient object and the method requires a persistent object. Because the object is not persistent, the method cannot do anything, so it need not be joined to the session.

- An unassociated thread calls a method that does not operate on persistent objects, for example, calls to `ObjectStore.getAutoOpenMode()` and `ObjectStore.setLazyWriteLocking`.
- An unassociated thread calls a method that has already been executed. The thread might join the session automatically if it executes the method anyway. For example, when an unassociated thread tries to open a database that is already open, PSE Pro joins the thread to the session that the database belongs to, even though the thread does not actually do anything.

Nonglobal sessions

In a nonglobal session, PSE Pro automatically joins threads to the session when the call from the thread implies that session. This means that the call specifies an argument that is already associated with that session. This includes the object on which the method is invoked.

After PSE Pro automatically joins a thread to a session,

- The thread is associated with the session until you remove it from the session or the session terminates.
- PSE Pro performs the called method.

Associating a Persistent Object with a Session

How does an object become associated with a session? It happens implicitly. Assume that a thread is already associated with a session. This associated thread successfully calls a PSE Pro API. If there are any objects that result from that call, PSE Pro associates them with the session that the calling thread belongs to.

As a result of explicit and implicit association, a session provides a context for a set of persistent objects and a set of PSE Pro API objects, such as a `Database` object and a `Transaction` object.

The session defines a namespace. The namespace defines unique names (and, consequently, identities) for databases, segments, clusters, transactions, and persistent objects. While it is possible for threads in different sessions to share objects, doing so is incorrect and usually results in exceptions.

If the thread in which an object was materialized leaves the session, the object remains associated with the session.

Rules for Joining a Thread to a Session Automatically

Because of the associations between objects and a particular session, some API calls imply a session. If there is no global session,

- A call that implies a session allows the calling thread to be joined to the implied session automatically.
- A call that does not imply a session does not allow the calling thread to be joined to a nonglobal session automatically.

If a thread associated with one session makes a call that implies another session, PSE Pro signals `WrongSessionException`.

Examples of Calls That Imply Sessions

A call that implies a session is a call that specifies an argument that is already associated with a session. It can also be a call in which the object on which the method is called is associated with a session. When these calls are in a thread that is not associated with a session, PSE Pro automatically joins the thread to the session with which the argument is already associated. It does not matter whether it is a global or nonglobal session. Some examples of API calls that imply a session follow:

- `Database.close()` — The `Database` argument was associated with a session when it was created.
- `ObjectStore.migrate(object, placement, export)` — The `placement` argument specifies a segment or database, which was associated with a session when it was initialized.
- `ObjectStore.destroy(object)` — The `object` argument designates a persistent object. It was associated with a session the first time it was accessed.

Examples of Calls That Do Not Imply Sessions

A call that does not imply a session is a call that does not specify an argument that is associated with a session. When these calls are in a thread that is not associated with a session, PSE Pro cannot join the thread to a session automatically if it is a nonglobal session. Examples of API calls that do not imply a session follow:

- `Database.open(name, openType)` — A static method. The `Database` object does not exist yet so the `name` argument is not associated with a session.
- `Transaction.begin(type)` — Another static method, and the `Transaction` object does not exist yet.
- `ObjectStore.majorRelease()` — Also a static method.
- A call that accesses a transient object.
- A call that never accesses a persistent object, for example, `ObjectStore.getAutoOpenType()` and `ObjectStore.setLazyWriteLocking()`.

Explicitly Associating Threads with a Session

To join a thread to a session explicitly, you call the `Session.join()` method.

Session.join()

To associate a thread with a session explicitly, call the `Session.join()` method. The method signature is

```
public void join()
```

This associates the current thread (the thread that contains the call to `join()`) with the session on which the `join()` method is called.

PSE Pro signals exceptions when joining threads to sessions in the following cases:

- When you try to join a thread to a session that has been terminated, PSE Pro signals `NoSessionException`.

- When you try to join a thread that already is in a session to another session, PSE Pro signals `WrongSessionException`.

However, if you try to join a thread to a session to which the thread already belongs, PSE Pro does nothing.

To join a thread to a session for a bounded duration of time, try something like the following:

```
Session session;
try {
  session.join();
  ...;
  ...;
  ...;
}  finally {
    session.leave();
}
```

Working with Threads

After you associate a thread with a session, it is important that you understand how to use the thread within the framework of a session. To that end, this section discusses

- Cooperating Threads
- Noncooperating Threads
- Synchronizing Threads
- Removing Threads from Sessions
- Threads That Create a Session
- Other Threads
- Threads and Applets on page 49
- Determining Whether PSE Pro Is Initialized for the Current Thread

Cooperating Threads

All threads associated with a particular session cooperate with each other. That is, they

- Share transactions, persistent objects, and locks on PSE Pro data
- View the same state of any databases they access

For example, suppose thread `A` and thread `B` are cooperating threads (that is, they belong to the same session). `A` and `B` are running asynchronously. Each thread is issuing a sequence of operations and these sequences are interleaved in an unpredictable manner.

For PSE Pro, it is as if these operations are all coming from the same thread. It does not matter which operation comes from `A` and which operation comes from `B`. PSE

Pro views the operations as being in a single sequence because they are issued from cooperating threads.

If A or B starts a transaction, it does not matter which thread issues the call. The transaction begins for both threads, regardless of the thread that actually starts the transaction. Any changes performed by A or B during the transaction are visible to both threads and can be acted on by either thread. Similarly, if A commits the transaction, it is just as if B commits the transaction. So B must be in a state in which it is all right to commit the transaction. A and B must cooperate.

Noncooperating Threads

Threads that do not belong to the same session cannot share transactions, persistent objects, or locks on data, and cannot view the same state of the database. Threads that belong to different sessions are noncooperating threads. With ObjectStore or PSE Pro, a different session can belong to the same process or a different process.

Two or more noncooperating threads can open the same database at the same time and access the same root object. If two or more noncooperating threads access the same object in the database, an equivalent number of distinct instances of the persistent object exist — one for each thread. The identity test, ==, does not show them to be identical.

Synchronizing Threads

Your application is responsible for synchronizing activity among cooperating threads when the transaction is committed or aborted. In general, your application must avoid accessing the database while a thread is committing the transaction and until a cooperating thread starts a new transaction. If a transaction is aborted, cooperating threads might need to retry database operations.

Additional information about synchronizing threads is in Multiple Cooperating Threads on page 87.

Removing Threads from Sessions

A thread can leave a session at any time, including while a transaction is in progress. This does not affect the transaction, nor any threads that are still joined to that session. With or without a transaction in progress, it is all right if no threads are associated with a session. The session does not terminate. A thread can join a session later to finish the transaction. If no thread does that, PSE Pro aborts the transaction when the session terminates.

To end the association of a thread with a session, call the `Session.leave()` method. The method signature is

```
public static void leave()
```

After you execute this method, the current thread is no longer joined to the session. If the current thread is already not associated with the session on which the method is called, PSE Pro signals `NoSessionException`.

If your application or PSE Pro shuts down a session, PSE Pro causes any associated threads to leave the session before it performs the shutdown.

If a thread is associated with a session and the thread terminates, it leaves the session automatically.

Threads That Create a Session

There is nothing special about the thread that creates a session. This thread can leave the session and any threads associated with that session can continue operating.

When your application calls the `Session.createGlobal()` or `Session.create()` method, PSE Pro does not associate the thread that calls the method with the newly created session. For that thread to join the new session, it must call the `Session.join()` method.

Other Threads

A thread that does not belong to a session cannot use the PSE Pro API. However, a thread need not be associated with a session to call `Session.isActive()` successfully.

If a session has a transaction in progress, a thread that is not associated with that session must not use persistent objects that belong to that session. See Threads and Persistent Objects on page 50.

If a session does not have a transaction in progress, any thread, including threads that do not belong to that session, can access persistent objects to the degree they were left visible when the application committed or aborted the transaction. See Ending a Transaction on page 83.

Threads and Applets

If you want to use PSE Pro with applets, it is likely that you will use cooperating threads. Typically, one thread performs initialization and another thread runs the applet's body.

Determining Whether PSE Pro Is Initialized for the Current Thread

You can use the `Session.getCurrent()` method to determine whether PSE Pro is initialized for the current thread. The method signature is

```
public static Session getCurrent()
```

This method returns the session with which the current thread is associated. If the current thread is not associated with a session, this method returns `null`.

Threads and Persistent Objects

Each persistent object is associated with exactly one session. Any modification to the state of a persistent object must be done by a thread that cooperates in the session to which the persistent object belongs.

After you terminate a session, the persistent objects and API objects that were associated with it when it was terminated continue to be associated with the terminated session. One exception to this is when you call the `Database.close()` method with a `true` argument. This causes the persistent objects to be retained as transient objects, which are not associated with any session.

The information in this section is provided to help you ensure that threads access the correct objects. This section discusses these topics:

- Multiple Representations of the Same Object
- Example of Multiple Sessions
- Application Responsibility
- Effects of Committing a Transaction
- Description of Allowable Simultaneous Actions on page 51
- API Objects and Sessions

Multiple Representations of the Same Object

When you have multiple sessions, it is possible to have multiple persistent objects that represent the same object in the database. For example, a thread belonging to session `A` accesses object `X`. Then a thread belonging to session `B` accesses object `X`. There are two persistent objects that represent `X`. Each is a representation of the same object in the database. If you use the `==` operator on session `A`'s `X` and session `B`'s `X`, the result is that they are not identical; they are not the same object. Within a session, PSE Pro preserves object identity.

Example of Multiple Sessions

The following example shows actions you can and cannot perform when you have multiple sessions. In this example, suppose you have `sessionA` and `sessionB` and

- `threadA` is associated with `sessionA`.
- `threadB` is associated with `sessionB`.

In `threadA`, you start a transaction and read the contents of a persistent object called `objectA`. Because `threadA` is associated with `sessionA`, `objectA` belongs to `sessionA`. You commit the transaction with `ObjectStore.RETAIN_UPDATE`.

At this point, in `threadB`, you can read or modify `objectA` unless a transaction is in progress in `sessionB`. However, any modifications are discarded when `sessionA` starts a transaction.

Application Responsibility

It is the responsibility of the application to ensure that noncooperating threads act on persistent objects only in the ways allowed when a transaction is not in progress.

If you have a Java static variable that contains a persistent object and there are two separate sessions, you must decide the session that owns the static variable. In other words, if there is a Java static variable whose value is a persistent object, that persistent object is associated with one session.

Effects of Committing a Transaction

When a thread commits a transaction, it affects only those persistent objects that belong to the same session that the thread belongs to.

Caution

You must ensure that an object never refers to an object that belongs to a different session. This is crucial because transitive persistence (performed when committing a transaction) must never reach an object that belongs to another session. If it does, PSE Pro signals `WrongSessionException`.

Array objects

When a thread commits a transaction, if PSE Pro reaches an object whose class does not implement `IPersistent`, PSE Pro treats the object as a transient object and migrates it to a database. This works correctly for immutable classes such as `Integer` and `String`. For array objects, this can cause unpredictable results because one session might modify the object while another session is using the old contents.

Description of Allowable Simultaneous Actions

Noncooperating threads can simultaneously have the same database open for read-only. So, in a sense, noncooperating threads can share a database even though they are not cooperating.

Also, noncooperating threads can simultaneously open different databases for update. However, noncooperating threads cannot simultaneously open the same database for update. It is as if they are distinct processes; if one thread has a database open for update and a noncooperating thread tries to open that database, either the second thread waits or PSE Pro throws `com.odi.DatabaseLockedException`. What happens depends on the type of the transaction.

See also Description of Concurrency Rules on page 87.

API Objects and Sessions

Each PSE Pro API object is related to one session. These metaobjects are

- `Cluster`
- `Database`
- `DatabaseRootEnumeration`
- `DatabaseSegmentEnumeration`
- `Segment`
- `Transaction`

If you open the same database from two noncooperating transactions, each session has its own `Database` object to represent the database. These `Database` objects are not identical; that is, `==` returns false.

If you try to use a database in the wrong session, PSE Pro signals `WrongSessionException`.

Description of PSE Pro Properties

When you create a session, you can specify a properties argument. This section provides the following information about this argument:

- About Property Lists Relevant to PSE Pro
- Description of com.odi.queryDebugLevel on page 54
- Description of com.odi.stringPoolSize
- Description of com.odi.trapUnregisteredType
- Description of com.odi.useDatabaseLocking on page 57
- Description of com.odi.useFsync on page 57
- Description of com.odi.useFsync

About Property Lists Relevant to PSE Pro

When you create a session, there are the following two relevant property lists:

- The `java.util.Properties` object that is the second argument to the method that creates the session
- The system property list

Finding the value of a property

To find the value of a property, PSE Pro checks the `java.util.Properties` object. If it provides a value, PSE Pro uses it. If it does not provide a value, PSE Pro checks the system property list.

Passing a property value

When you want to pass a property value to the method that creates a session, you typically put it in the `java.util.Properties` object that is an argument to the method that creates the session.

There is only one system property list for each Java VM. Multiple sessions in the same Java VM all use the same system property list. For more information about system properties, see `System.getProperty`, in section 21.6 of the *Java Language Specification*.

Defining a system property

All PSE Pro property names start with `com.odi`. You can pass in property information by defining it as a system property. For example:

```
Properties props = System.getProperties();
props.put("com.odi.useDatabaseLocking", "true");
Session session = Session.create(null,props);
```

There is also a `System.setProperties()` method that resets the `System` property list.

The JDK allows you to specify a system property by including

`-D`*`parameter=value`*

on the `java` command line before the class name. Each such specification defines one system property. Not all Java virtual machines run this way.

Defining a Properties object

If you want to construct your own property list, the type of the property list argument is `java.util.Properties`. For example:

```
Properties props = new Properties();
props.put("com.odi.useDatabaseLocking", "true");
Session session = Session.create(null,props);
```

Description of com.odi.disableCrossTransactionCaching

The `com.odi.disableCrossTransactionCaching` property defaults to `true`. This means that PSE Pro makes retained objects hollow at the start of a new transaction. This is the same way that ObjectStore functions.

When this property is set to `false`, at the beginning of a new transaction, PSE Pro does not hollow objects whose contents were retained after a committed transaction. This means that after you commit a transaction with `ObjectStore.RETAIN_READONLY` or `ObjectStore.RETAIN_UPDATE`, you have access to all objects that you read or updated during the previous transaction. If you then start a new transaction, the contents of objects that were retained are not hollowed out. If you always retain object contents when you commit a transaction, you have access to all objects you read or updated in the current session.

A disadvantage of setting this property to false is that there is no comparable property in ObjectStore. If your application depends on the behavior allowed when `disableCrossTransactionCaching` is set to `false`, it will not work with ObjectStore.

This property will be deprecated in a future release when an API that performs a similar function is available.

Description of com.odi.disableWeakReferences

The `com.odi.disableWeakReferences` property defaults to `false`. This means that PSE Pro uses the weak reference facility of the JDK. If you set this property to `true`, it disables the weak reference facility and PSE Pro does not use it.

When you start the first session in a Java process, the setting of the `com.odi.disableWeakReferences` property is in effect for the duration of the Java process. If you terminate the session and start another session with a different value for the `com.odi.disableWeakReferences` property, the new value is ignored.

A weak reference to an object is a reference that does not prevent the object from being garbage collected by the Java VM's garbage collector. PSE Pro uses weak references in its internal object table to hold references to unmodified persistent objects. If your program does not have any references to persistent objects and the reference in the object table is the only reference, the object can be garbage collected. If the persistent object has been modified and the changes have not yet been saved, PSE Pro uses a strong reference, which does not allow the object to be garbage collected.

Description of com.odi.queryDebugLevel

The `com.odi.queryDebugLevel` property allows you to control debugging output when you are using the query facility defined in the `com.odi.util.query.Query` class. The default level is `0`, which does not output any information. Set the property to a value greater than `0` to print debugging output to `System.err`. As the value of this property increases, the query facility outputs more information as follows:

- `0` — No debugging information.
- `1` — Indexes that are required and that would be useful; statistics about indexed and nonindexed lookups.
- `5` — Query parser tree and query evaluation tree.
- `10` — Information about generated classes.

Description of com.odi.stringPoolSize

The `com.odi.stringPoolSize` property allows you to specify the number of newly created strings PSE Pro maintains in the string pool for the current session. In PSE Pro, the default is `"100"`.

When PSE Pro is about to migrate a string into a segment, it first checks the string pool for an identical string in the same segment. If it finds one, ObjectStore uses the string that is already stored in the segment instead of adding a new identical string to the segment. The information about the strings that are available to be shared is maintained only for the current transaction. The strings that are available to be shared are maintained in a string pool. PSE Pro resets the string pool to empty at the start of each transaction.

For example, suppose you create two instances of a `Person` object in a transaction. In each instance, the value of the `name` field is `Lee`. If you store both instances in the database in the same transaction, and in the same segment, PSE Pro adds only one instance of the string `"Lee"` to the database. This is true even though the Java VM might contain two instances of the string `"Lee"`. When PSE Pro writes the first `"Lee"` string in the segment, ObjectStore notes it in the string pool. Before PSE Pro stores the next instance of `"Lee"` in the segment, it checks the string pool to see if an identical instance is already in the segment. However, if the two `Person` objects were stored in different segments, the two instances of the string `"Lee"` would be migrated to the database and stored separately in each segment.

Continuing the example, suppose you use two transactions and you store one instance of `Person` in each transaction. The result is that there are two identical `"Lee"` strings in the segment. This is because PSE Pro resets the string pool to be empty at the start of each transaction. Consequently, PSE Pro cannot reuse the `"Lee"` string from the previous transaction.

Caution

If you use `ObjectStore.destroy()` to destroy strings explicitly, you might want to turn off string pooling so you do not inadvertently destroy a string that is shared by different objects. Alternatively, you can use the persistent GC to reclaim strings when they are no longer referenced. Using the GC is usually preferable to explicitly calling `destroy()`, because it is safer to let the persistent GC collect unreachable strings. Also, this approach is often more efficient and results in less database fragmentation.

Description of com.odi.trapUnregisteredType

The `com.odi.trapUnregisteredType` property is useful for troubleshooting `ClassCastException`s. The default is that this property is not set and it is usually best to use the default.

When PSE Pro encounters an object of a type for which it does not have information (that is, the type is unregistered), it checks the setting of the `com.odi.trapUnregisteredType` property.

If the property is not set, PSE Pro creates an instance of the `UnregisteredType` class to represent the object of the unknown type. Your application continues to run as long as it does not try to use the `UnregisteredType` object. Often, this can work well because your application has no need for that particular field. However, if you do try to use the object of the unregistered type, PSE Pro signals `ClassCastException`.

If `com.odi.trapUnregisteredType` is set, PSE Pro does not create an `UnregisteredType` object. Instead, it signals `FatalApplicationException` and

provides a message that indicates the name of the unregistered class. For additional information, see Handling Unregistered Types on page 133.

Description of **com.odi.useDatabaseLocking**

The `com.odi.useDatabaseLocking` property allows you to turn cross-process locking off. By default, cross-process locking is turned on. This means that if a session has a database open, a session in another Java VM process cannot open that database. If a session tries to open a database that is already open in another process, PSE Pro throws `com.odi.DatabaseLockedException`.

The `com.odi.useDatabaseLocking` property is a PSE Pro feature; it is not a Java interface to ObjectStore feature. If you are also using the Java interface to ObjectStore, beware of confusing this property with the ObjectStore java interface `Database.acquireLock()` API. This method allows a session to explicitly lock a particular database for exclusive use.

Description **of com.odi.useFsync**

Set the `com.odi.useFsync` property to `false` if you do *not* want PSE Pro to flush (write) data to disk from the operating system's buffers prior to committing the transaction. The default value for this property is `true`.

This means that PSE Pro flushes all modified data to disk before a `Transaction.commit()` call returns. This does not interfere with the ability to roll back changes if a transaction aborts. When this property is set to `false`, there are fewer disk writes so performance can be faster. However, there is the risk of corrupted data if the machine is not shut down cleanly. Even if the Java VM is shut down cleanly, the operating system must flush its buffers cleanly as well.

Description of com.odi.useImmediateStrings

When you set the `com.odi.useImmediateStrings` property to `true`, `String`s that are eight characters (bytes) or less are stored as immediate values when a session is created. This means that these `String`s are not stored as independent objects in the database, resulting in a smaller database and avoiding the run-time overhead for tracking the `String`s in the object table.

When `com.odi.useImmediateStrings` is enabled, `String`s that are stored as values of fields in persistent objects are not persistent, unless you explicitly call `ObjectStore.migrate` on the `String`s to make them persistent.

By default, the `com.odi.useImmediateStrings` property is enabled (set to `true`).

Chapter 4
Managing Databases

You create databases to store your objects. The `Database` class provides the API for creating and managing databases.

Contents

This chapter discusses the following topics:

Note

Release 7.0 changes the database format. If you have Release 6.*x* databases and you want to use them with Release 7.0, you must upgrade those databases with the `osjup70` utility. See osjup70: Upgrading Databases to 7.0 Format on page 283. After you upgrade databases to Release 7.0, you cannot use them with Release 6.*x* applications.

Creating a Database

The `Database` class is an abstract class that represents a database. When you create a database,

- There must be an active session, or PSE Pro signals `NoSessionException`.
- A transaction must not be in progress, or PSE Pro signals `TransactionInProgressException`.

Databases are cross-platform compatible. You can create databases on any supported platform and access them from any supported platform.

This section discusses the following topics:

Method Signature for Creating a Database

To create a database, call the static create method on the `Database` class and specify the database name and an access mode. The method signature is

```
public static Database create(String name, int fileMode)
```

PSE Pro signals `AccessViolationException` if the access mode does not provide owner write access. If the access mode provides owner write access, PSE Pro ignores any other specified permissions. This is due to a limitation in the Java implementation.

Example of Creating a Database

For example:

```
import com.odi.*;
class DbTest {
  void test() {
    Database db = Database.create("objectsrus.odb",
      ObjectStore.OWNER_WRITE);
    ...
  }
}
```

This example creates an instance of `Database` and stores a reference to the instance in the variable named `db`. The `Database.create` method is called with two parameters.

The first parameter specifies the pathname of a file.When you are using PSE Pro, you must end this name with `.odb`.

The second parameter specifies the access mode for the database.

Terminology note

`Database` is an abstract class, so PSE Pro actually creates an instance of a subclass that extends `Database`. From your point of view, it does not matter whether PSE Pro creates an instance of `Database` or an instance of a `Database` subclass.

Result of Creating a Database

The result is a database named `"objectsrus.odb"` with an access mode that allows the owner to modify the database. The example stores the reference to the `Database` object in the `db` variable. This means that `db` represents, or is a handle for, the `objectsrus.odb` database.

For each database you create, PSE Pro creates an instance of `Database` to represent your database. Each database is associated with one instance of `Database`.

Specifying a Database Name in Creation Method

When you create or open a database you must specify or pass in a name of the form

database_name.odb

PSE Pro performs standard I/O to the host system to create or open the database in the location you specify. The operating system interprets the database name in the context of the local file system. Before you specify a directory that is physically remote, you must make the directory appear to be local by using NFS, or some other network file system.

For each database you create, PSE Pro creates three files.

- The first file has the pathname you specify or pass when you create the database. The pathname must be unique and must end with .odb.
- The second file has the same name except that PSE Pro replaces the .odb extension with an .odt extension.
- The third file also has the same name and PSE Pro replaces the .odb extension with an .odf extension.

The .odb, .odt, and .odf files together make up the database.

The PSE Pro API uses the *path*.odb name. Your application should never specify the *path*.odt or *path*.odf name. But you must be aware of the .odt and .odf files so that you can correctly move or copy a database.

When the Database Already Exists

If you try to create a database that already exists, PSE Pro signals DatabaseAlreadyExistsException. Before you create a database, you might want to check to see whether it exists and destroy it if it does. For example, you can insert the following before you create a database:

```
try {
  Database.open(dbName, ObjectStore.UPDATE).destroy();
}  catch(DatabaseNotFoundException e) {
}
```

Warning

Do this only if you want to destroy and recreate your database. Otherwise, invoke Database.open().

Segments

PSE Pro creates each database with one segment that contains one cluster, which is a variable-sized region of disk space.

The segment is the smallest unit of storage that can be garbage collected, because objects in the same segment can freely reference each other. However, objects in different segments, and even in different databases, can also refer to each other. Such cross-segment references are explicitly tracked in the exported object table. As a result, a single segment can be garbage collected without inspecting the other segments it might refer to or the other segments that might refer to it.

Initially, the size of the segment is about 3 KB, and it consists of a single cluster. As you store additional objects in the segment, PSE Pro increases the size of the segment automatically.

You cannot create additional segments.

Note

For 32-bit platforms, all transient C++ peer objects reside on the same transient cluster. However, for 64-bit platforms, transient C++ peer objects might reside on different transient clusters. Therefore, on 64-bit platforms, when you call `Cluster.of()` on two different transient C++ peer objects, different clusters might be returned because a different `Cluster` object is returned for each 2^{32} byte range of transient pointers.

After you create an iterator for the objects in a cluster, other sessions are blocked from destroying that cluster until you end your transaction.

If you create the iterator and then destroy the cluster, the next call to `next()` or `hasNext()` causes PSE Pro to signal `ClusterNotFoundException`.

There is a bug that makes the `iterator` work incorrectly if you call `Transaction.checkpoint(RETAIN_STALE)`. Doing so causes the next use of the iterator to signal `ObjectException` because of stale objects. This will be fixed in a future release.

Determining Whether a Database, Segment, or Cluster Is Transient

Sometimes there are Java peer objects that identify C++ objects that have been transiently allocated. PSE Pro stores these C++ objects in the transient database, transient segment, and transient cluster. To determine whether a database or segment is transient, you can do the following:

```
CPlusPlus.getTransientCluster() == cluster
CPlusPlus.getTransientSegment() == segment
CPlusPlus.getTransientDatabase() == database
```

Only Java peer objects are stored in the transient database, segment, or cluster. Java primary objects are never stored in the transient database, transient segment, or transient cluster, even if they are transient.

If you try to retrieve the cluster, segment, or database of a transient primary object, PSE Pro signals `ObjectNotPersistentException`.

Opening and Closing a Database

A database can be either open or closed. You must open a database before you can store or access objects in that database. When an application opens a database, it does not matter whether

- A transaction is in progress.
- The database is already open.

When an application closes a database, a transaction cannot be in progress and the database must be open.

This section discusses the following topics:

Opening a Database

When you open a database, it does not matter whether a transaction is in progress, nor does it matter whether the database is already open.

When you create a database, PSE Pro creates and opens the database. To open an existing database, call the static `Database.open()` method. The method signature is

```
public static Database open(String name, int openMode)
```

For example:

```
Database db = Database.open("myDb.odb", ObjectStore.READONLY);
```

The first parameter specifies the pathname of your database. The second parameter indicates the open mode of the database.

Possible Open Modes

PSE Pro provides constants that you can specify for the `openMode` parameter to `Database.open()`. The constants you can specify for `openMode` are

- `ObjectStore.UPDATE` to read and modify a database.
- `ObjectStore.READONLY` to read but not modify a database.
-

Note

Using `Database.open(name, openMode)` might block the first time you open a database in a session. You can avoid this blocking by opening the database the first

time using the `MVCC` mode, closing the database, then using the `Database` object to reopen the database in the mode you want.

Incorrect attempts to modify

If you open a database with `ObjectStore.READONLY`and attempt to modify an object, PSE Pro signals `UpdateReadOnlyException` when you try to commit the transaction.

Concurrency

PSE Pro grants read and update locks when you start a transaction (if there is an open database) or when you open a database in a transaction. Consequently, multiple sessions can have the same database open for read and update at the same time. They cannot, however, actually read and update the same database at the same time. For detailed information about concurrency control see Description of Concurrency Rules on page 87.

Example

Suppose you previously created and closed a database that is represented by an instance of a `Database` subclass stored in the `db` variable. You can call the instance `open()` method to open your database in the following way:

```
db.open(ObjectStore.READONLY);
```

You can use the static class `open()` method in the following way:

```
db = Database.open("myDb.odb", ObjectStore.READONLY);
```

Typically, both lines cause the same result. However, they might cause different results if a database has been destroyed and recreated.

Database recovery

If you try to open a database with read-only access and the database needs to be recovered, PSE Pro throws `com.odi.DatabaseNeedsRecoveryException` and the open operation fails.

To recover a database that is in an inconsistent state due to a system crash, open it for update. This automatically recovers the database, if necessary. Alternatively, you can run the `osjcheckdb` utility with the `-openUpdateForRecovery` option.

Threads, Sessions, and Open Databases

A PSE Pro application that uses one thread or one group of cooperating threads uses only one session. Such an application can have only one database open at a time. The application must close the database before it can open another one.

In PSE Pro, multiple sessions can open the same database. Each session has its own PSE Pro object table, which includes a copy of each object that is associated with the session. This means that using multiple sessions can consume more transient memory than if threads cooperate in the same session. Concurrency rules determine whether or not two sessions can actually access the same database at the same time.

A PSE Pro application that uses multiple threads that do not cooperate with each other has one session for each noncooperating thread. If your application uses multiple groups of cooperating threads in which the groups do not cooperate with each other, it has one session for each separate group of cooperating threads. An application can use the different sessions to open the same database or different databases. Each session must close its database before it can open another one.

A single PSE Pro application can have more than one database open when it uses multiple sessions. A single session cannot have more than one database open.

Opening the Same Database Multiple Times

After a database is initially opened, if threads in the same session subsequently open the database again, the same database object is returned. For example:

```
db1 = Database.open("foo", ObjectStore.UPDATE);
db2 = Database.open("foo", ObjectStore.UPDATE);
```

The expression `db1 == db2` returns true. They refer to the same database object. Consequently, a call to `db1.close()` or `db2.close()` closes the same database. No matter how many times you open a database, a single call to the `close()` method closes the database.

Closing a Database

To close a database, call the `close()` method on the instance of the `Database` subclass that represents the database, for example:

```
db.close();
```

The database must be open and there must not be a transaction in progress when `Database.close()` is called; otherwise, an exception is thrown.

Object state after close

When you close a database, all persistent objects that belong to that database become stale or transient. If the last committed transaction that operated on the database retained persistent objects, you can use an overloading of `close()` that allows you to specify what should happen to the retained objects. (For information about retained objects, see Committing Transactions to Save Modifications on page 111.) The method signature is

```
public void close(boolean retainAsTransient)
```

Specify `true` to make retained objects transient. If you specify `false`, it is the same as calling the `close()` method without an argument. All access to retained objects ends.

Suppose you close a database and make retained objects transient. In the next transaction, if you reread an object from the database that you retained as a transient object, you then have two separate copies of the same object: One is transient and one is persistent. You do not have two references to a single object. When you close a database, all object identity is gone. After you close a database, the database is still associated with the session in which it was closed.

If you do not close

If you do not close a database, PSE Pro closes it when you shut down PSE Pro.

If you are using PSE Pro, the `open()` and `close()` operations are very efficient. Multiple opens and closes should not be a performance drain.

Releasing process lock

When your application opens a database, PSE Pro creates a directory that functions as a lock against other Java VM processes. The name of the directory is

database_name`.odx`

PSE Pro maintains this directory in the same directory as your database. When your application closes the database or terminates the session that opened the database, PSE Pro deletes the `.odx` directory, which releases the lock.

Your application must close the database or terminate the session to allow PSE Pro to release the lock. Consequently, ObjectStore Technical Support recommends that you call the `Database.close()` or `Session.terminate()` method in the `finally` clause of your program. If you do not close the database, it remains locked and other VM processes cannot access it. For an example see Preventing More Than One Process from Accessing a Database on page 90.

Database identity

Within a session, PSE Pro maintains database identity even after you close a database. For example, consider the following code:

```
import com.odi.*;
public class Goo {
  public static void main(String[] args) {
    Session session = Session.create(null, null);
    session.join();
    try {
      try {
        Database db = Database.create("my.odb", 0664);
        db.close();
      } catch (DatabaseAlreadyExistsException e) {
      }
      Database db1 = Database.open("my.odb",
        ObjectStore.READONLY);
      db1.close();
      Database db2 = Database.open("my.odb",
        ObjectStore.READONLY);
      System.out.println(db1 == db2);
    } finally {
      session.terminate();
    }
  }
}
```

If you run a program with the previous code, the system displays `true`.

When Closing a Database Is Required

An application must close a database

- Before it can open some other database in the same session.
- To permit another application or another session to modify the database.
- To delete the `.odx` directory so that another process can access the database. See Preventing More Than One Process from Accessing a Database on page 90.

Shutting Down PSE Pro Closes Open Databases

You can call the `close()` method on the database, or you can ensure that your application calls `Session.terminate()` or `ObjectStore.shutdown()`, which automatically closes an open database. Use `try/finally` in the main entrypoint to make sure the shutdown occurs if there is an unexpected exception. See Importance of closing the database on page 90.

Objects in Closed Databases

Objects in a closed database are not accessible. However, if you close a database with an argument of `true`, PSE Pro retains the persistent objects as transient objects.

Moving or Copying a Database

You can move or copy a database provided that

- You keep the `.odb`, `.odt`, and `.odf` files together in the same directory.
- The pathname is the same for the `.odb`, `.odt`, and `.odf` files.
- The database is not open for update.

When you copy or move a database, you do not need to copy or move the `odx` directory, if it exists.

You can move or copy databases among different supported platforms.

If you want to copy the objects in a database to another database, you can follow these steps:

1. Open the source database.
2. Start a transaction.
3. Call `ObjectStore.deepFetch()` to obtain the objects you want to copy.
4. Commit the transaction with `ObjectStore.RETAIN_READONLY`.
5. Close the database with `ObjectStore.RETAIN_TRANSIENT`.
6. Open the target database.
7. Begin an update transaction.
8. Save the transient data some place in the target database.

Performing Garbage Collection in a Database

The PSE Pro persistent garbage collector (GC) collects unreferenced Java objects in a PSE Pro database. Persistent garbage collection frees storage associated with objects that are unreachable. It does not move remaining objects to make the free space contiguous.

Restrictions

You cannot use the PSE Pro for Java persistent GC on old C++ PSE databases.

Contents

This section discusses the following topics:

Background About the Persistent Garbage Collector

The PSE Pro persistent GC is independent of the Java VM GC. The Java VM GC is strictly a transient object garbage collector. It never operates on objects in the database.

When the persistent GC runs, it starts a transaction and locks the database for update. When the database is locked for update, PSE Pro performs database locking differently depending on whether other concurrent sessions are running in the same Java VM or in different Java VMs.

If the sessions are running concurrently in the same Java VM, then any of the sessions can have the PSE Pro database open as long as there are no transactions running in the sessions, when the persistent GC has locked the database for update. By running multiple sessions in the same Java VM, you can interleave transactions with persistent garbage collection.

If the sessions are running concurrently in different Java VMs, then none of the sessions can have the PSE Pro database open when the persistent GC has locked the database for update.

The GC performs its job in two major phases. In the mark phase, the GC identifies the unreachable objects. In the sweep phase, the GC frees the storage used by the unreachable objects.

You can specify a segment or a database to be garbage collected. Since each PSE Pro database contains only one segment, there is no difference between using the segment GC API and the database GC API.

It is usually best to avoid destroying strings (or objects) altogether and let the persistent GC take care of destroying such unreachable objects. The persistent GC typically can destroy and reclaim such objects very efficiently, because it can batch such operations and cluster them effectively. If you set up the GC to run when the system is lightly loaded, you can effectively defer the overhead of the destroy operations to a time when your system would otherwise be idle, thus getting greater real throughput from your application when you really need it.

API for Collecting Garbage in a Database

To perform garbage collection on a database, call the `Database.GC()` method. This method invokes the `Segment.GC()` method on each segment in the database. The method signature is

```
public java.util.Properties GC(java.util.Properties GCproperties)
```

For the *`GCproperties`* parameter, specify null or a `Properties` object for the garbage collection operation. The properties are described in the next section, as they are the same for `Segment.GC()`. If the `GCproperties` parameter is null, ObjectStore uses the default properties as defined in the documentation for `Segment.GC()`. The properties you can specify are the same as the properties for `Segment.GC()`.

API for Collecting Garbage in a Segment

To perform garbage collection on a segment, call the `Segment.GC()` method. The signature is

```
public java.util.Properties GC(
)
```

A transaction must not be in progress for the current session. The database that contains the segment you want to garbage collect must not be open for the current session. However, you must open it to create a `Database` object to represent it. After you close the database you want to garbage collect, you can call `Database.GC()` on the `Database` object that represents your closed database.

The `GC()` method returns a `Properties` object that contains information about the results of the garbage collection. The properties in this object are as follows:

- `com.odi.gc.reclaimedObjects` is the number of objects that were collected by the GC operation.
- `com.odi.gc.reachableObjects` is the number of objects that the GC found to be reachable. This property is always null for PSE Pro. It is meaningful only for ObjectStore.

Command-Line Utility for Collecting Garbage

The command-line utility for collecting garbage in a database is `osjgcdb`.

.

Schema Evolution: Modifying Class Definitions of Objects in a Database

You can modify the class definitions for objects already stored in a database. This process is called schema evolution, because a database schema is a description of the classes whose instances are stored in a database.

The technique for performing schema evolution is to use serialization with a dump/load utility. This section provides an example of how to do this.

The topics discussed in this section are

- When is schema evolution required?
- Considerations for using serialization to evolve schema
- Steps for using sample code that uses serialization with a dump/load utility
- Sample code

When Is Schema Evolution Required?

If you change your class in one of the following ways, you must evolve the schema:

- Add or remove a persistent instance field

- Change the type of a persistent instance field
- Change the order of persistent instance fields

hashCode()

Also, you might need to perform schema evolution if you add or remove the `hashCode()` method. If you use the postprocessor, it determines whether to add a `hashCode()` method. If it previously added a `hashCode()` method and now it does not, or if it previously did not add a `hashCode()` method and now it does, schema evolution is required.

Inheritance

You cannot use schema evolution to change the inheritance hierarchy of a class by adding, removing, or changing a superclass.

Allowed changes

You can make the following changes to your class and you are not required to evolve the schema:

- Add or remove class or instance methods
- Add or remove class fields
- Add or remove transient instance fields
- Add or remove an implementation of an interface

It is always advisable to make a copy of your database before you evolve its schema. This allows you to restore the database if there are errors.

Considerations for Using Serialization to Perform Schema Evolution

To evolve a schema using serialization, use the following steps:

1. Use serialization to dump the contents of a database.
2. Modify your class definitions.
3. Reload the data.

The next two sections provide instructions for using a sample program that evolves a schema using serialization. Before you use this sample program, you should be aware of the issues described in this section.

Serialize objects

To serialize objects into the database, the classes of all the objects stored in the database must implement `java.io.Serializable`. If you have a database that contains objects that do not implement `Serializable`, you can modify the class definitions just to implement `Serializable`, recompile them, and still access the database. This allows you to dump the database to a file before you make the real class modifications.

readObject()

When you modify a class after doing the dump, you must ensure that the `readObject()` method considers the old and new versions of the class to be compatible. The most straightforward way to do this is to create a `static final long` field called `serialVersionUID` in the modified class. This field must have the same value as the serial version UID for the original class. You can obtain the value for the original class with the `serialver` utility, for example:

```
serialver DumpReload$LinkedList
DumpReload$LinkedList: static final long serialVersionUID =
```

```
-5286408624542393371L;
```

Database size

The database whose schema you want to evolve must be small enough to fit into heap space. If it is not, you must customize the code that dumps and loads the database. You would have to organize your data so that you do not have to serialize all the data in the database at one time.

Large numbers of connected objects

The use of `ObjectStore.deepFetch()` is a performance concern for very large object graphs. The current implementation of `deepFetch()` is not careful about bounding stack space. A consequence of this is that it is sometimes impossible to successfully perform the `deepFetch()` operation for very large object graphs.

Steps for Using Sample Schema Evolution Serialization Code

The sample program that is provided in the next section takes an argument that causes it to perform one of the following three actions:

- Create a database with some data in it, such as instances of `OSHashtable`, `OSVector`, or linked lists
- Use object serialization to dump data in the database to a file
- Use object serialization to reload the data from the file

For example, you can use the sample program to add a new field to the `LinkedList` class. To do so, follow these steps:

1 Place the code in a file called `DumpReload.java`.

2 Set your `CLASSPATH` environment variable to include the directory that contains the `osjcfpout` file and the `DumpReload.java` file.

3 Compile the program with the command

```
javac DumpReload.java
```

4 Run the postprocessor to annotate the `DumpReload` and `LinkedList` classes:

```
osjcfp -dest osjcfpout DumpReload.class \
DumpReload$LinkedList.class
```

5 Create the database:

```
java DumpReload create data.odb
```

6 Use serialization to dump the data:

```
java DumpReload dump data.odb data.out
```

7 Change the `LinkedList` class. Do this by removing the comment flag from the `newField` field in `LinkedList`.

8 Recompile the class:

```
javac DumpReload.java
```

9 Rerun the postprocessor to annotate the `DumpReload` and `LinkedList` classes:

```
osjcfp -dest osjcfpout DumpReload.class \
DumpReload$LinkedList.class
```

10 Use serialization to reload the data:

```
java DumpReload reload data.odb data.out
```

Sample Code for Using Serialization to Perform Schema Evolution

Following is the sample program that uses serialization to perform schema evolution:

```
import com.odi.*;
import com.odi.util.OSHashtable;
import com.odi.util.OSVector;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import java.util.Enumeration;

public class DumpReload {
  static public void main(String argv[]) throws Exception {
    if (argv.length >= 2) {
      if (argv[0].equalsIgnoreCase("create")) {
        createDatabase(argv[1]);
      } else if (argv[0].equalsIgnoreCase("dump")) {
        dumpDatabase(argv[1], argv[2]);
      } else if (argv[0].equalsIgnoreCase("reload")) {
        reloadDatabase(argv[1], argv[2]);
      } else {
        usage();
      }
    } else {
      usage();
    }
  }

  static void usage() {
    System.err.println(
      "Usage: java DumpReload OPERATION ARGS...\n" +
      "Operations:\n" +
      "   create DB\n" +
      "   dump FROMDB TOFILE\n" +
      "   reload TODB FROMFILE\n");
    System.exit(1);
  }

  /* Create a database with 3 roots. Each root contains an
    OSHashtable of OSVectors that contain some Strings. */

  static void createDatabase(String dbName) throws Exception {
    ObjectStore.initialize(null, null);

    try {
      Database.open(dbName, ObjectStore.UPDATE).destroy();
    } catch (DatabaseNotFoundException DNFE) {
    }
    Database db = Database.create(dbName, 0644);

    Transaction t = Transaction.begin(ObjectStore.UPDATE);
    for (int i = 0; i < 3; i++) {
      OSHashtable ht = new OSHashtable();
```

```
    for (int j = 0; j < 5; j++) {
      OSVector vec = new OSVector(5);
      for (int k = 0; k < 5; k++)
        vec.addElement(new LinkedList(i));
      ht.put(new Integer(j), vec);
    }
    db.createRoot("Root" + Integer.toString(i), ht);
  }
  t.commit();
  db.close();
}

static void dumpDatabase(String dbName, String dumpName)
  throws Exception {
  ObjectStore.initialize(null, null);

  Database db = Database.open(
        dbName, ObjectStore.READONLY);

  FileOutputStream fos = new FileOutputStream(dumpName);
  ObjectOutputStream out = new ObjectOutputStream(fos);

  Transaction t = Transaction.begin(ObjectStore.READONLY);

  /* Count the roots and write out the count. */
  Enumeration roots = db.getRoots();
  int nRoots = 0;
  while (roots.hasMoreElements()) {
    String rootName= (String) roots.nextElement();
    /* Skip internal OSJI header */
    if (!rootName.equals("_DMA_Database_header")) nRoots++;
  }
  out.writeObject(new Integer(nRoots));

  /* Rescan and write out the data.
    The deepFetch() call is necessary because it obtains the
    contents of all objects that are reachable from root,
    and makes them available for serialization. */
  roots = db.getRoots();
  while (roots.hasMoreElements()) {
    String rootName = (String) roots.nextElement();
    if (!rootName.equals("_DMA_Database_header")) {
      out.writeObject(rootName);
      Object root = db.getRoot(rootName);
      ObjectStore.deepFetch(root);
      out.writeObject(root);
  }

  t.commit();

  out.close();
}

static void reloadDatabase(String dbName, String dumpName)
  throws Exception {
  ObjectStore.initialize(null, null);

  try {
    Database.open(
      dbName, ObjectStore.UPDATE).destroy();
```

```
    } catch (DatabaseNotFoundException DNFE) {
    }
    Database db = Database.create(dbName, 0644);

    FileInputStream fis = new FileInputStream(dumpName);
    ObjectInputStream in = new ObjectInputStream(fis);

    Transaction t = Transaction.begin(ObjectStore.UPDATE);

    int nRoots = ((Integer) in.readObject()).intValue();
    while (nRoots-- > 0) {
      String rootName = (String) in.readObject();
      Object rootValue = in.readObject();
      System.out.println("Creating "+rootName+" "+ rootValue);
      db.createRoot(rootName, rootValue);
    }

    t.commit();
    db.close();
  }

  static
  class LinkedList implements java.io.Serializable {
    private int value;
    private LinkedList next;
    private LinkedList prev;
    //private Object newField;
    static final long serialVersionUID= -5286408624542393371L;

    LinkedList(int value) {
      this.value = value;
      this.next = null;
      this.prev = null;
    }
  }
}
```

Destroying a Database

When you destroy a database, PSE Pro makes all objects in the database permanently inaccessible and deletes the `.odb`, `.odt`, and `.odf` files.

You cannot recover a destroyed database except from backups.

To destroy a database, call the `destroy()` method on the `Database` subclass instance, for example:

```
db.destroy();
```

The database must be open for update and a transaction cannot be in progress.

When you destroy a database, all persistent objects that belong to that database become stale.

Obtaining Information About a Database

You can call methods on a database to answer the following questions:

- Is a Database Open?
- What Kind of Access Is Allowed?
- What Is the Pathname of a Database?
- What Is the Size of a Database?
- With Which Session Is the Database or Segment Associated?
- Which Objects Are in the Database?
- Are There Invalid References in the Database?

Is a Database Open?

To determine whether a database is open, call the `isOpen()` method on the database, for example:

```
db.isOpen();
```

This expression returns `true` if the database is open. It returns `false` if the database is closed or if it was destroyed. To determine whether `false` indicates a closed or destroyed database, try to open the database.

What Kind of Access Is Allowed?

To check what kind of access is allowed for an open database, call the `getOpenMode()` method on the database. The database must be open or PSE Pro signals `DatabaseNotOpenException`. The method signature is

```
public int getOpenMode()
```

This method returns one of the following constants:

- `ObjectStore.READONLY`
- `ObjectStore.UPDATE`

Following is an example of how you can use this method:

```
void checkUpdate(Database db) {
  if (db.getOpenMode() != ObjectStore.UPDATE)
    throw new Error("The database must be open for update.");
}
```

What Is the Pathname of a Database?

To find out the pathname of a database, call the `getPath()` method on the database, for example:

```
String myString = db.getPath();
```

What Is the Size of a Database?

To obtain the size of a database, call the `getSizeInBytes()` method on the database. The database must be open and a transaction must be in progress, for example:

```
db = Database.open("myDb.odb", ObjectStore.READONLY);
Transaction tr = Transaction.begin(ObjectStore.READONLY);
long dbSize = db.getSizeInBytes();
```

This method does not necessarily return the exact number of bytes that the database uses. The value returned might be the result of your operating system's rounding up to a block size. You should be aware of how your operating system handles operations such as these.

With Which Session Is the Database or Segment Associated?

To obtain the session with which a database or segment is associated, call the `Placement.getSession()` method. The method signature is

```
public Session Placement.getSession()
```

Which Objects Are in the Database?

The `osjshowdb` utility displays information about one or more databases. This utility is useful when you want to know how many and what types of objects are in a database. You can use this utility to verify the general contents of the database.

Information about the `osjshowdb` utility is in osjshowdb: Displaying Information About a Database on page 282.

Are There Invalid References in the Database?

The `osjcheckdb` utility or the `Database.check()` method checks the references in a database. This tool scans a database and checks that there are no references to destroyed objects. The most likely cause of dangling references is an incorrectly written program. You can fix dangling references by finding the objects that contain them and overwriting the invalid references with something else, such as a null value. In addition to finding references to destroyed objects, the tool performs various consistency checks on the database.

Information about the `osjcheckdb` utility is in osjcheckdb: Checking References in a Database on page 280.

Exported objects are not retained on a commit, checkpoint, or abort when the `ObjectStore.RETAIN_STALE` argument is specified. Instead, use external references or the `ObjectStore.RETAIN_HOLLOW` argument value.

Database Operations and Transactions

For each database operation, there are rules about whether it can be performed

- Inside a transaction

- Outside a transaction
- Both inside and outside a transaction

The following table shows the rules that apply to certain operations. If your application tries to perform a database operation that breaks a rule, you receive a run-time exception.

Database Operation	*Can Be Performed Inside/Outside Transaction?*	*Database Open?*
`acquireLock()`	Inside	Open
`check()`	Inside	Open
`close()`	Outside	Open
`create()`	Outside	Not applicable
`createRoot()`	Inside	Open
`destroy()`	Outside	Open
`destroyRoot()`	Inside	Open
`GC()`	Outside	Closed
`getDefaultSegment()`	Inside	Open
`getOpenMode()`	Both	Open
`getPath()`	Both	Open
`getRoot()`	Inside	Open
`getRoots()`	Inside	Open
`getSegment()`	Inside	Open
`getSegments()`	Inside	Open
`getSizeInBytes()`	Inside	Open
`isOpen()`	Both	Open or closed
`of()`	Inside	Open
`open()`	Both	Open or closed
`setDefaultSegment()`	Inside	Open
`setRoot()`	Inside	Open
`show()`	Inside	Open

Restrictions on Databases

The `fileMode` argument to `com.odi.Database.create()` does not work because Java does not allow you to specify the access mode. However, the file mode must still specify that the owner can modify the file.

PSE Pro does not allow cross-database references.

Controlling Database Size

You might notice that the `.odt` file associated with your database is much larger than it should be for the number of stored objects. In this situation, run the `osjshowdb` utility to find out the number of destroyed objects.

When you destroy an object in a database, doing so frees the space. However, the database file does not shrink in size. One strategy for reducing the actual file size of your database is to use object serialization:

1. Stream out all database roots.
2. Create a new database.
3. Stream in the database roots from the old database.
4. Destroy the old database.

You can also use this strategy to evolve your classes.

Chapter 5
Working with Transactions

A transaction is a logical unit of work that runs in a session. Only one transaction can be running in a session at a time. A transaction is a consistent and reliable portion of the execution of a program.

In your code, you place calls to the PSE Pro API to mark the beginnings and ends of transactions. Initial access to a persistent object must always take place inside a transaction. Depending on how the transaction is committed, additional access to persistent objects might be possible.

Either the database is updated with all of a transaction's changes to persistent objects or the database is not updated at all. If a failure occurs in the middle of a transaction, or you decide to abort the transaction, the contents of the database remain unchanged.

A transaction can obtain a write lock or a read-only lock on the database opened by the session. A write lock prevents other sessions in the Java VM from writing to the database. A read-only lock allows other sessions to access the database by using read-only transactions.

Contents

This chapter discusses the following topics:

Starting a Transaction

PSE Pro provides the `com.odi.Transaction` class to represent a transaction. You should not make subclasses of this class.

This section discusses the following topics:

- Calling the begin() Method
- Description of Transaction Types on page 80
- Allowing Objects to Be Modified in a Transaction
- Difference Between Update and Read-Only Transactions

Calling the begin() Method

To start a transaction, call the `begin()` method on the `Transaction` class. This returns an instance of `Transaction` and you can assign it to a variable. The method signature is

Method signature

```
public static Transaction begin(int type)
```

The transaction type determines whether PSE Pro waits for a write lock on adatabase. There can be only one write lock on a database, but there can be multiple read-only locks on a database. The type of the transaction can be `ObjectStore.UPDATE` or `ObjectStore.READONLY`.

If there is no open database when you start the current transaction, PSE Pro tries to obtain a read-only lock or a write lock as soon as the session tries to open a database, depending on the type of transaction.

Example

```
Transaction tr = Transaction.begin(ObjectStore.UPDATE);
```

This example returns a `Transaction` object that represents the transaction just started. The result is stored in `tr`. This is an update transaction, which means that the application can modify database contents.

Description of Transaction Types

The `type` of the transaction can be one of the following:

Update

- `ObjectStore.UPDATE` allows the application to modify data in the open database in the current transaction. PSE Pro immediately obtains a write lock for the open database if an exclusive write lock is available. If the database is locked by another session for either read or write, PSE Pro waits indefinitely until the database is no longer locked, and grants the write lock when the database is available.

Read-only

- `ObjectStore.READONLY` lets the application read but not modify data in the open database in the current transaction. PSE Pro immediately obtains a shared read lock on the open database if a read lock is available. If the open database is locked by another session for read access and there are no sessions waiting for an update lock, PSE Pro grants a read lock. If the database is locked or being waited for by another session for update, PSE Pro waits until the database is no longer locked for write, and grants the read lock when the database becomes available.

Nonblocking update

- `ObjectStore.UPDATE_NON_BLOCKING` allows the application to modify data in the open database in the current transaction. PSE Pro immediately obtains a write lock on the open database if the write lock is available. If the open database is locked by another session, PSE Pro throws `com.odi.DatabaseLockedException`, and the transaction begin operation fails.

Nonblocking read-only

- `ObjectStore.READONLY_NON_BLOCKING` allows the application to read but not modify data in the open database in the current transaction. PSE Pro immediately obtains a read lock on the open database if a read lock is available. If the database is locked by another session for read access and there are no sessions waiting for a write lock, PSE Pro grants a read lock. If the database is locked for write or being waited for by another session for update, PSE Pro throws `com.odi.DatabaseLockedException` and the transaction begin operation fails.

Allowing Objects to Be Modified in a Transaction

To modify persistent objects, you must specify the transaction type to be `ObjectStore.UPDATE`. Also, any database you modify must have been opened for update. Note that even if you open a database for read-only, PSE Pro allows you to start an update transaction. An application does not receive an exception until it tries to modify persistent objects inside the read-only database.

If you try to modify persistent data in a read-only transaction, PSE Pro signals `UpdateReadOnlyException`.

Difference Between Update and Read-Only Transactions

You can start a transaction for `READONLY` or for `UPDATE`. The major difference between the two transaction types is that when you start a transaction for `READONLY`, PSE Pro performs additional checks during the transaction and when you commit the transaction. These checks ensure that changes are not saved in the database if they were made in a read-only transaction. There is no difference in performance between a read-only transaction and an update transaction.

Working Inside a Transaction

A transaction is associated with the session that is associated with the thread that starts the transaction. A transaction remains active until you explicitly commit it or until it aborts. A session can have only one active transaction. Concurrent transactions must be in separate sessions.

This section discusses the following topics:

PSE Pro and Transactions

PSE Pro allows multiple sessions. Noncooperating threads (separate sessions) in the same Java VM process can simultaneously have separate read-only transactions against the same database. Also, noncooperating threads (sessions) in the same Java VM can simultaneously have separate update transactions against different databases; each noncooperating thread (session) must update a different database. Only one session at a time can open an update transaction against a particular database. Cooperating threads can update the same database. See Cooperating Threads on page 47. Also see Noncooperating Threads on page 48.

Obtaining the Session Associated with the Current Transaction

The current session is the session that a thread most recently joined. To obtain the session that is associated with the current transaction, call the `Transaction.getSession()` method. The method signature is

```
public Session Transaction.getSession()
```

To obtain the transaction that is associated with the current session, call the `Session.currentTransaction()` method. The method signature is

```
public Transaction Session.currentTransaction()
```

Is a Transaction in Progress for the Current Session?

To determine whether there is a transaction in progress for the current session, call the `Transaction.inTransaction()` method on `Transaction`. The method signature is

```
public static boolean inTransaction()
```

This method returns `true` if there is a transaction in progress for the current session. Otherwise, it returns `false`. It is worth noting that `inTransaction()` returns `false` if the calling thread is not joined to the current session. This can be important if you use an unassociated thread to check whether there is a transaction, and then try to close the database based on a false response. The previously unassociated thread would automatically be joined to the session to close the database. If a transaction is actually in progress, PSE Pro signals `TransactionInProgressException`, which is, of course, unexpected because `inTransaction()` returned `false`.

Transaction Already in Progress

Nested transactions are not allowed. If you try to start a transaction when a transaction for the current session is already in progress, PSE Pro signals `TransactionInProgressException`.

Obtaining Transaction Objects

An application can obtain the transaction object for the current thread by calling the static `current()` method on the `Transaction` class. The method signature is

```
public static Transaction current()
```

This method returns the transaction object associated with the current session, for example:

```
Transaction.current().commit()
```

This example commits the current transaction. If no transaction is in progress, `current()` signals `NoTransactionInProgressException`.

Ending a Transaction

When transactions terminate successfully, they commit and their changes to persistent objects are saved in the database. When transactions terminate unsuccessfully, they abort and their changes to persistent objects are discarded.

For read-only transactions, there are no advantages to committing them rather than aborting them, nor to aborting them rather than committing them.

This section discusses the following topics:

Committing Transactions

PSE Pro provides the `Transaction.commit()` method for ending a transaction successfully. When an application commits a transaction, PSE Pro

- Saves and commits any changes in the database
- Performs transitive persistence if applicable (see page 28)
- Sets the state of persistent objects that were accessed or referenced in the transaction
- Releases the lock it held on the database being accessed

Transitive persistence

When PSE Pro commits a transaction, it checks to see if there are any transient objects that are referred to by persistent objects. If there are, and if all referred-to objects are persistence-capable objects, PSE Pro stores the referred-to objects in the database. This is the process of transitive persistence. If any referred-to object is not persistence capable, PSE Pro signals `ObjectNotPersistenceCapableException`.

Caution

You must ensure that an object never refers to an object that belongs to a different session. This is crucial because transitive persistence must never reach an object that belongs to another session.

Making objects stale

To commit a transaction and make the state of persistent objects stale, call the `commit()` method with no argument. For example:

```
tr.commit();
```

The method signature is

```
public void commit()
```

Setting object state

To commit a transaction and be flexible about the state of persistent objects after the transaction, call the `commit(retain)` method on the transaction. The values you can specify for `retain` are described in Committing Transactions to Save Modifications on page 111. The method signature is

```
public void commit(int retain)
```

The following example commits the transaction and specifies that the contents of the active persistent objects should remain available to be read.

```
tr.commit(ObjectStore.RETAIN_READONLY);
```

What Can Cause a Transaction Commit to Fail?

When PSE Pro tries to commit a transaction, if PSE Pro encounters any of the situations in the following list, it causes the transaction commit to fail. When PSE Pro aborts a transaction commit, it signals `AbortException`.

- A persistent object references an object that is not persistence capable.
- A persistent object was updated to reference a stale object.

When an Application Terminates During a Commit Operation

If an application terminates at any time other than during a `Transaction.commit()` operation, PSE Pro returns the database to the state it was in before the transaction started.

If an application terminates during a call to `Transaction.commit()`

- If PSE Pro has the `com.odi.useFsync` system property set to `true`, which is the default, it is always able to recover the database, even if the operating system crashes during a commit.
- If the `com.odi.useFsync` property is set to `false`, PSE Pro is able to recover the database if the operating system flushed the database to disk prior to the crash.

A Control-C or a system crash can cause an application to terminate during a call to `Transaction.commit()`.

Aborting Transactions

PSE Pro provides the `Transaction.abort()` method for ending a transaction unsuccessfully. An abort can happen explicitly through the `Transaction.abort()` method or implicitly because a session is terminated or there is a system exception. When an application aborts a transaction, PSE Pro

- Ensures that the objects in the database are as they were just before the aborted transaction started
- Sets the state of persistent objects from the transaction
- Returns any transient objects that were made persistent during the transaction to their transient state
- Releases the lock on the database

Transient objects

Only the state of the database is rolled back. The state of transient objects is not undone automatically. For example, if you created new transient objects during the transaction, they still exist after the transaction aborts. Applications are responsible for undoing the states of transient objects. Any form of output that occurred before the abort cannot be undone.

Open databases

If you opened a database during the transaction, PSE Pro keeps it open. If a database was open before the aborted transaction was started, it remains open after the abort operation.

Application failure

If an application fails during a transaction, when you restart the application the database is as it was before the transaction started. If an application fails during a transaction commit, when you restart the application, either the database is as it was before the transaction that was being committed or the database reflects all the transaction's changes. This depends on how far along in the commit process the application was when it terminated. Either all or none of the transaction's changes are in the database.

abort()

To abort a transaction and set the state of persistent objects to the state specified by `Transaction.setDefaultAbortRetain()`, call the `abort()`. method. The default state is stale. The method signature is

```
public void abort()
```

For example:

```
tr.abort();
```

abort(retain)

To abort a transaction and specify a particular state for persistent objects after the transaction, call the `abort(retain)` method on the transaction. The values you can specify for *`retain`* are described in Specifying a Particular State for Persistent Objects on page 124. The method signature is

```
public void abort(int retain)
```

The following example aborts the transaction and specifies that the contents of the active persistent objects should remain available to be read.

```
tr.abort(ObjectStore.RETAIN_READONLY);
```

Has This Transaction Been Aborted?

To determine whether or not a transaction has been aborted, call the `Transaction.isAborted()` method on the transaction. The method signature is

```
public boolean isAborted()
```

This method returns true if the transaction has been aborted. It returns false for any other status. PSE Pro does not require that the calling thread belong to a session. Also, the session in which the transaction was started is not required to be active.

Is This Transaction Active?

To determine whether or not a transaction is active, call the `Transaction.isActive()` method on the transaction. The method signature is

```
public boolean isActive()
```

This method returns true if the transaction has not been aborted or committed. It returns false if it has been aborted or committed. PSE Pro does not require that the calling thread belong to a session. Also, the session in which the transaction was started is not required to be active.

Determining Transaction Boundaries

When determining whether to commit a transaction, consider database state, and interdependencies among cooperating threads.

Inconsistent Database State

You should not commit a transaction if the database is in a logically inconsistent state. A database is considered to be in an inconsistent state if at that moment a just-started transaction would encounter problems on viewing the current state of the data.

Consider your database to be something that moves from one consistent state to another. You should commit a transaction only when the state is consistent. When is a database consistent? If you start your application at this very moment, is the database completely usable exactly the way it is now?

For example, suppose your database contains information about married couples. Couples refer to one another through a `spouse` field. At a particular moment, suppose a person in the database refers to another person in the database through its `spouse` field, but that spouse does not refer to the first person. At that moment, the database is in an inconsistent state.

Another inconsistency you should avoid is migrating objects into a database that are not reachable from any root. Such objects become unreachable if the application fails between transactions or they are removed inappropriately from the database when the garbage collector runs.

When the database state is consistent, you might decide not to commit the transaction. However, if you do not commit, you risk losing changes if PSE Pro aborts the transaction. You should always commit changes before you inform a user or another interface that a particular task was accomplished.

Multiple Cooperating Threads

If your application uses cooperating threads, you must take this into account when determining when to commit transactions. For example, you do not want to create a situation in which one thread commits a transaction while a cooperating thread is updating persistent objects. The `commit()` method might make all persistent objects stale for all cooperating threads. If the `commit()` method retains persistent objects, PSE Pro discards any modifications to retained persistent objects at the start of the next transaction. You must coordinate the `Transaction.begin()` and `Transaction.commit()` operations among cooperating threads.

Synchronizing threads is like having a joint checking account. Suppose the amount in the checking account is $100.00. Your partner writes a check for $50.00. Then you try to cash a check for $75.00. This does not work. It does not matter that it was your partner and not you who wrote the check for $50.00. You and your partner have to cooperate.

Performance Considerations

Committing a transaction, even a read-only transaction, has a certain amount of overhead associated with it. If you have a lot of small transactions, you might want to combine some of them into larger transactions.

Description of Concurrency Rules

PSE Pro controls concurrent access to a database between sessions within the same Java VM. At any given time, there can be multiple reader sessions or *one* writer session for a database. An application must obtain a lock on a database before it can access the database. For each database, PSE Pro provides multiple read-only locks or one write lock. PSE Pro provides concurrency control between processes (Java VMs) by preventing multiple processes from opening the same database at the same time.

PSE Pro grants and releases database locks at transaction boundaries. Each session can have at most exactly one active transaction and one open database at a time. Typically, your application acquires a lock when it calls `Transaction.begin()` and releases a lock when it calls `Transaction.commit()` or `Transaction.abort()`. The result is that a database is either single-session or read-only. The following topics provide more details on the concurrency rules:

Definition of One Writer and Multiple Readers

One writer means that within a process, only one session can have an update transaction in progress for a particular database. Two cooperating threads (remember that cooperating threads are always in the same session) that are both updating a database count as one writer. Two noncooperating threads in the same process (in other words, two threads from two different sessions in the same process) cannot both have an update transaction in progress against the same database at the same time.

Multiple readers means that multiple sessions in the same Java VM can have active read-only transactions against the same database at the same time. If one or more sessions in a Java VM has a read-only transaction in progress for a database, no other session can have an update transaction in progress for the same database.

Description of Database Locks

A read-only lock allows the session that holds it to read, but not modify, the database. A write lock allows the session that holds it to modify the database.

For a particular database at a given point in time, PSE Pro grants either multiple read-only locks or one write lock. If a session has a read-only lock, other sessions in the same Java VM can also have read-only locks, but no session can have a write lock. If a session has a write lock, no other session can have a lock on that database.

When PSE Pro Grants Database Locks

PSE Pro grants a lock on a database when it starts a transaction or when it opens a database inside a transaction. Your application can open a database inside or outside a transaction.

If you open a database inside a transaction, and if the lock you want is available, PSE Pro immediately grants the lock. PSE Pro releases the lock when you end the transaction in which you acquired the lock.

If you open a database outside a transaction, PSE Pro does not grant a lock on the database until you start a transaction in the same session. This means that multiple sessions can have the database open at the same time. Some can have it open for read-only and others for update. However, not one of these sessions can access the objects in the opened database. To access objects, a session must start a transaction and obtain a lock on the opened database.

Determining If a Lock Is Available

For a session to access a database, PSE Pro must grant the session a read-only or update lock on the database. If there are no locks on a database, multiple read-only locks are available or one update lock is available.

Read locks

If one or more sessions have a read-only lock on a database, other sessions in the same Java VM can obtain read-only locks on that database. However, suppose a session is waiting for a write lock on the same database. In this situation, PSE Pro does not grant any new read-only locks. Instead, PSE Pro does one of the following:

- Allows the session that wants the read-only lock to wait for all read-locks to be released, and then wait for the update lock to be released
- Throws `com.odi.DatabaseLockedException` because the lock that is wanted is not available

What PSE Pro does depends on the transaction type you specify when you start the transaction in which the lock is wanted. If you specify `ObjectStore.READONLY` or `ObjectStore.UPDATE`, PSE Pro allows the session to wait. If you specify `ObjectStore.READONLY_NON_BLOCKING` or `ObjectStore.UPDATE_NON_BLOCKING`, PSE Pro throws `com.odi.DatabaseLockedException`.

Update lock

If a session has a write lock on a database, and another session tries to obtain a lock on the same database, no lock is available. Again, depending on the transaction type, PSE Pro allows the session to wait or throws `com.odi.DatabaseLockedException`.

Lock queue

PSE Pro maintains a queue of sessions that want locks for a particular database. There is no limit to the number of entries in the queue. PSE Pro handles the lock requests in the order in which they arrive.

Effects of Concurrency Rules

With PSE Pro, concurrency within a single Java VM process is allowed as follows:

- PSE Pro permits multiple threads within a Java VM to cooperate in the same session and transaction, and read or write a single database.
- PSE Pro permits multiple sessions within a Java VM to have independent active read-only transactions against the same database at the same time.
- PSE Pro permits threads that belong to different sessions in the same Java VM to start their own independent transactions and use them to read the same database or read or write different databases.
- PSE Pro does not permit different sessions to have independent transactions that read and write to the same database at the same time.

It is possible for multiple sessions in the same Java VM to have the same database open for update at the same time. This can happen when none or one of the sessions has an update transaction in progress. Also, multiple sessions can open the database for update and then start read-only transactions.

A session can never have more than one database open at a time. If you need multiple databases open at the same time, you must use a session for each database that needs to be simultaneously open.

Preventing More Than One Process from Accessing a Database

The default behavior of PSE Pro is that concurrent access from different Java VM processes to the same PSE Pro database is not allowed. When your application opens a database, PSE Pro creates a `.odx` directory in the same directory as your database and uses the `.odx` directory to lock out other processes.

If your application opens a database from one process, and another process tries to open the same database, PSE Pro throws the `com.odi.DatabaseLockedException`. It does not matter whether the first process opened the database for read-only or update. Nor does it matter whether the second process tries to open the database for read-only or update.

This means that two sessions in two different Java VM processes cannot access the same database at the same time.

The **.odx** directory

When your application opens a database, PSE Pro creates a directory that functions as a lock against other Java VM processes. The name of the directory is

`database_name``.odx`

PSE Pro maintains this directory in the same directory as your database. When your application closes the database or terminates the session that opened the database, PSE Pro deletes the `.odx` directory, which releases the lock. The `.odx` directory does not contain any files. This directory acts as a lock to the other Java VMs.

Importance of closing the database

Your application must close the database or terminate the session to allow PSE Pro to release the lock. Consequently, ObjectStore recommends that you call the `Session.terminate()` method in a `finally` clause of a `try` statement. (Terminating a session closes any open databases.) If you do not close the database, it remains locked and other VM processes cannot access it. Here is an example of terminating the session:

```
Session session = Session.create(null,null);
session.join();
try {
  Database db = Database.open("foo.odb", ObjectStore.UPDATE);
  // ... do something ...
} finally {
   session.terminate();
}
```

If a database open fails, then the variable representing the database is null. If you call `Database.close()` on a null object, PSE Pro throws `NullPointerException`.

If a process exits without closing a database or without terminating the session that opened the database, PSE Pro does not delete the `.odx` directory. This can happen, for example, if an application does not call `Database.close()` or `Session.terminate()`, or if there is a crash, `System.exit()` call, or Control-C exit. In such situations, the `.odx` directory continues to exist and incorrectly blocks the database from being opened.

DatabaseLocked Exception

If a process tries to open a database and PSE Pro throws `com.odi.DatabaseLockedException` because of the presence of the `.odx` directory, it can mean either of these things:

- Another Java VM process has the database open. Consequently, no other process can open the database. You must wait until the process closes the database.
- The `.odx` directory is left over from a session that was not correctly terminated, and no Java VM process is using the database. Verify that no process is actually using the database. Then you can delete the `.odx` directory and try again to open the database.

The text of the `com.odi.DatabaseLockedException` is something like this:

```
com.odi.DatabaseLockedException:
Unable to obtain the lock for the database named
"/tmp/mumble.odb". The database might be in use by another
process, or some other error might
have occurred during the attempt to create the "/tmp/mumble.odx"
directory. Check for other processes on any hosts that have access
to the file system that contains this database. If some other process
is using the database, wait for it to complete before trying to access
it again. If no other process is accessing the database, you can clear the lock by
removing the directory named "/tmp/mumble.odx".
at com.odi.imp.pro.RandomAccessFile.<init>(RandomAccessFile.java:89)
at
com.odi.imp.pro.ObjectTable$SharedCachedRandomAccessFile.<init>(ObjectTable.java:278)
at com.odi.imp.pro.ObjectTable$CachedRandomAccessFile.<init>(ObjectTable.java:3312)
at com.odi.imp.pro.ObjectTable$CachedObjectTable.<init>(ObjectTable.java:3563)
at com.odi.imp.pro.ObjectTable.openDatabase(ObjectTable.java:1637)
at com.odi.imp.pro.Database.serverOpenDatabaseByName(Database.java)
at com.odi.imp.pro.Server.serverOpenDatabase(Server.java:145)
at com.odi.imp.Server.openDatabase(Server.java:198)
at com.odi.imp.Database.open(Database.java)
at com.odi.tools.ShowDB.main(ShowDB.java)
```

Changing default behavior

If you do not want your application to have the default behavior, you can set the `com.odi.useDatabaseLocking` system property to `false`. By default, this property is set to `true`. You can set the property to false by calling the `Session.create()`, or `Session.createGlobal()` method. However, ObjectStore Technical Support does not recommend that you do this.

If you turn off cross-process database locking, PSE Pro cannot prevent multiple processes from accessing the same database at the same time. Concurrent modifications could corrupt your user and system data.

Without default database locking turned on, you must set up your own cross-process concurrency control mechanism. In your application, you must lock databases against concurrent updates from multiple processes.

Chapter 6
Storing, Retrieving, and Updating Objects

This chapter provides information about how to store data in a database and how to read it back and update it. An application can access persistent data only inside a transaction and only when the database is open.

Contents

This chapter discusses the following topics:

Storing Objects

PSE Pro's Java API preserves the automatic storage management semantics of Java. Objects become persistent when they are referenced by other persistent objects. This is called persistence by reachability or transitive persistence. The application defines persistent roots and, when it commits a transaction, PSE Pro finds all objects reachable from persistent roots and stores them in the database.

To store objects in a database, do the following:

1. Open the database or create the database in which you want to store the objects. Be sure the database is opened for update. See Creating a Database on page 59.
2. Start an update transaction. See Starting a Transaction on page 79.
3. Create a database root or access an existing database root and specify that it refers directly or indirectly to one of the objects you want to store. See Working with Database Roots on page 96.
4. Commit the transaction. This stores the object that the database root refers to and any objects that object references. See Committing Transactions on page 83.

In general, you should not create a root for each object you want to store in a database. You must create at least one root to store an object in a database by which all other objects can ultimately be reached.

How Objects Become Persistent

Objects can become persistent in several ways:

- An application assigns a transient object to a database root. PSE Pro immediately migrates the object to the default cluster of the default segment in the database. When the transaction commits, any transient objects that are reachable from the object assigned to the root are also stored in the default segment and cluster.
- A transient object is reachable from a persistent object. When the transaction commits, PSE Pro stores the reachable object in the same segment and cluster as the persistent object.
- An application invokes the `ObjectStore.migrate()` method on a transient object and specifies a particular database, segment, or cluster. With PSE Pro, there is only one cluster and one segment in each database in which you can store objects. (ObjectStore allows multiple clusters and segments.)

What Is Reachability?

An object `B` is considered to be reachable from object `A` when `A` contains a reference to `B`, except when the reference is from a variable marked with Java's `transient` keyword. `B` is also reachable from `A` when `A` contains a reference to some object and that object contains a reference to `B`. There are no limits to levels of reachability.

Storing Java-Supplied Objects

Some Java-supplied classes are persistence capable, while others are not persistence capable and cannot be made persistence capable. A third category of classes can be made persistence capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 261.

Retrieving Persistent Objects

To read the contents of an object in a database, you must first obtain a reference to an object. There are several ways you can do this:

- Use a database root.
- Get the one database cluster and iterate over its objects
- Use external references.

The following sections describe these alternatives.

This section discusses the following topics:

- Steps for Retrieving Persistent Objects
- Determining the Database That Contains an Object
- Determining Whether an Object Has Been Stored

Steps for Retrieving Persistent Objects

Follow these steps to retrieve a persistent object from a database:

1. Open the database.
2. Start a transaction. If you want to modify the object, start an update transaction.
3. Obtain a persistent by object using a database root, an external reference, or by iterating over the objects in a segment or cluster of the database.
4. Access the object just as you would access a transient object.

   If you do not plan to run the postprocessor, see Making Object Contents Accessible on page 211.

Determining the Database That Contains an Object

You can use the `Database.of()` method to determine the database in which an object is stored. The method signature is

```
public static Database of(Object object)
```

If the specified object has been stored in a database, PSE Pro returns the database in which it is stored.

Determining Whether an Object Has Been Stored

To determine whether an object has already been stored in a database, call the `ObjectStore.isPersistent()` method. The method signature is

```
public static boolean isPersistent(Object object)
```

If the specified object has been stored in a database, PSE Pro returns `true`. The specified object must not be a stale persistent object. If it is, PSE Pro signals `ObjectException`.

Working with Database Roots

A root is a reference to an individual object. You always need at least one root in a database. You can get by with a single root, but you might find it convenient to have more. In general, you do not want every object in the database to be associated with a root. This is bad for performance. Each root refers to exactly one object. More than one root can refer to the same object. You cannot navigate backward from the referenced object to the database root.

This section discusses the following topics:

- Creating Database Roots
- Retrieving Root Objects
- Roots with Null Values
- Using Primitive Values as Roots
- Changing the Object Referred to by a Database Root
- Destroying a Database Root
- Destroying the Object Referred to by a Database Root
- How Many Roots Are Needed in a Database?

Creating Database Roots

When you create a database root, you give it a name and you assign an object to it. The database root refers to that object, and your application can use the root name to access that object. In other words, the object that you assign to a root is the value of that root. The database root and the object assigned to the root are two distinct objects.

You must create a database root inside a transaction. Call the `Database.createRoot()` method on the database in which you want to create the root. The method signature for this instance method on `Database` is

```
public void createRoot(String name, Object object)
```

The name you specify for the root must be unique in the database. If it is not unique, `DatabaseRootAlreadyExistsException` is signaled. The object that you specify to be referred to by the root can be either transient and persistence capable, or persistent (including null). If it is not yet persistent, PSE Pro immediately makes it persistent.

Example

Suppose you create the variable `db` to be a handle to a database opened for update, and an object called `anObject`, and you start an update transaction. The following line creates a database root:

```
db.createRoot("My Root Name", anObject);
```

Results

In the database referred to by `db`, this creates a database root named `"MyRootName"` and specifies that it refers to `anObject`. PSE Pro immediately stores `anObject` in the database referred to by `db`. When the transaction commits, PSE Pro stores in the database referred to by `db` any objects that are reachable from `anObject`, if they are not already in the database. If `anObject` or any object it references refers to any transient objects that are not persistence capable and you try to commit the transaction, PSE Pro signals `ObjectNotPersistenceCapableException`.

Multiple roots for one object

More than one root can reference the same object; an object can be associated with more than one root. For example:

```
db.createRoot("Root1", anObject);
db.createRoot("Root2", anObject);
```

Retrieving Root Objects

When you retrieve a root object, you obtain a reference to the object that is the value for the root. For example, suppose you assign an `OSVector` object, `myOSVector`, to a root named `myOSVectorRoot`. When you get the value of `myOSVectorRoot` by using the `Database.getRoot()` method, you receive a reference to the `OSVector` as follows:

```
OSVector myOSVector = (OSVector)db.getRoot("myOSVectorRoot");
```

Note that PSE Pro does not fetch the entire contents of the `OSVector` until you actually need it. You can obtain a reference to any object in a vector. For example, to obtain a reference to the fifth element in the vector `myOSVector`, use an assignment statement like the following:

```
Object fifth = myOSVector.elementAt(5);
```

PSE Pro fetches only enough contents from `myOSVector` so that it can return the reference to the fifth element. This means that PSE Pro has not yet fetched the contents of the elements in the vector. PSE Pro fetches the data for an element in a vector only when you try to access its contents.

If you develop your application by using the ObjectStore class file postprocessor, this delayed fetching is usually automatic. For more information on using the postprocessor, see Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171.

In some cases, you might want to force PSE Pro to prefetch an entire graph of connected objects even though the application does not explicitly refer to each object in the graph. Use the `ObjectStore.deepFetch()` method to do this.

List of all roots

To obtain a list of the roots in a database, call the `getRoots()` method on the database. The signature of this method is

```
public DatabaseRootEnumeration getRoots()
```

Roots with Null Values

It is possible to create a root with a `null` value. This is useful for creating roots in preparation for assigning objects to them later. If you create a root with `null` or later set a root to `null`, the `getRoot()` method returns a `null` value, which indicates that there is no object associated with the root. It does *not* mean that the root does not exist. If the root does not exist, PSE Pro signals `DatabaseRootNotFoundException`.

Using Primitive Values as Roots

If you want to store a primitive value as an independent persistent object, such as the value of a root, use an instance of a wrapper class, such as an `Integer`. For example:

```
db.createRoot("foo", new Integer(5));
```

This assigns the value `5` to the root named `foo`. You cannot directly store primitive values in a database. You can define a primitive value as a field in a persistence-capable object.

Changing the Object Referred to by a Database Root

After you create a database root, you can change the object that it refers to. Inside an update transaction, call the `Database.setRoot()` method on the database that contains the root. You must specify an existing root and you can specify either a transient (but persistence-capable) or a persistent object. The method signature for changing the object associated with a root is as follows. If PSE Pro cannot find the specified root, `DatabaseRootNotFoundException` is signaled.

```
public void setRoot (String name, Object object)
```

Destroying a Database Root

To destroy a database root, call the `destroyRoot()` method on the database that contains the root that you want to destroy. An update transaction must be in progress. Specify the name of the root. If PSE Pro cannot find the specified root, it signals `DatabaseRootNotFoundException`. The method signature is

```
public void destroyRoot (String name)
```

This has no effect on the referenced object except that it is no longer accessible from that root. It might still be the value of another root, or it might be pointed to by some other persistent object. If a value of a root is no longer referenced after the root is destroyed, the object becomes unreachable. You should invoke `ObjectStore.destroy()` on it while you still have a reference to it. See Performing Garbage Collection in a Database on page 67.

Destroying the Object Referred to by a Database Root

If you want to destroy the object that a database root refers to and you want to continue to use that database root, you also must set the root to refer to `null` or to another object. If you do not do this, you can retrieve the root, but if you try to use it PSE Pro throws `ObjectNotFoundException`. For example, the correct sequence is similar to the following:

```
Object object = db.getRoot("username");
db.setRoot("username", null);
ObjectStore.destroy(object);
```

How Many Roots Are Needed in a Database?

It is important to realize that you need not create a root for most objects that you want to store in a database. You need to create roots only for top-level objects that you want to look up by name. You must have at least one root to be able to navigate through a database. Without a root, you have no way of accessing the objects in the database.

Think of a database root as the root of a tree. From the root, you can climb the tree. For many applications, a root is some kind of container, such as an instance of `OSTreeMapString` or `OSVector`, or an array. After you create one or more database roots, you create other objects that are referred to by fields of the objects that the roots refer to. These objects become persistent when you commit the transaction in which you create them. In a subsequent transaction, you can look up the root objects by name and navigate from them to any other reachable persistent objects.

Too many roots can cause performance problems. The maximum practical number of roots within a database is between 1000 and 10,000. Databases store roots in an instance of `OSHashtable`.

How Many Objects Can You Store in a Database?

Each object you store in a database has the following overhead:

- Sixteen bytes disk space in the `.odt` file
- Four bytes disk space in the `.odb` file

When you are counting objects, primitive types do not count as objects. The `Long` and `Double` wrapper objects do count as objects. But, `Integer`, `Float`, and `Byte` do not count as objects. When you want to determine how many objects are in a database, be sure to account for any objects that are created by objects of classes that you define. For example, if you create an instance of `com.odi.util.OSVector`, you create more than one object because an `OSVector` has internal objects.

The overhead for the `.odf` file is approximately 4 KB for each 256 KB in the `.odb` file. Two gigabytes is the maximum size of a `.odb` file, which means that the maximum amount of data that a database can contain is two gigabytes.

PSE Pro has been designed to scale well for hundreds of thousands of objects and thousands of megabytes of data. Practical limits on database size and numbers of persistent objects vary according to your application and system. PSE Pro has delivered good performance when tested with loads of one million objects and one gigabyte of data. After PSE Pro opens a database, it fetches data as needed. It does not read the whole `.odt` file into memory as PSE Pro does.

Iterating Through the Objects in a Cluster, Segment, or Database

To obtain an iterator for the objects in a segment, call the `Segment.getObjects()` method. To obtain an iterator for the objects in a cluster, call the `Cluster.getObjects()` method. With PSE Pro these methods are equivalent, because there is only one segment containing one cluster in each database.

These methods give you access to any objects that are unreachable but that have not yet been garbage collected. The methods also provide an application-independent means of processing all objects within a segment or cluster. The method signature for both methods is

```
public Iterator getObjects()
```

This method returns an iterator object. After you have this object, you can use the following methods to iterate through the objects:

- `Iterator.next()`
- `Iterator.hasNext()`

The `Segment.getObjects()` and `Cluster.getObjects()` methods have an overloading that takes a `java.lang.Class` object as an argument and returns an iterator over all objects of that type in the database. The type can be an interface, a class, or an array type.

If your session or another session adds an object to a segment or cluster after you create an iterator, the iterator might or might not include the new object. If it is important for the iterator to include all objects accurately, you should create the iterator again.

After you create an iterator by using `Segment.getObjects()` and `Cluster.getObjects()`, objects in the segment or cluster might be destroyed. When you use an iterator to iterate through the objects, PSE Pro skips any destroyed objects. Note that this means that the `hasNext()` method might change the state of the iterator to skip destroyed objects.

You can use an iterator returned by the `getObjects()` method across transactions. If a transaction in which you use the iterator aborts, the iterator becomes stale and can no longer be used.

After you create an iterator for the objects in a segment or cluster, other sessions are blocked from destroying that segment or cluster until you end your transaction. If you create the iterator and then destroy the segment or cluster, the next call to `next()` or `hasNext()` causes PSE Pro to signal `SegmentNotFoundException` or `ClusterNotFoundException` respectively.

Using External References to Stored Objects

Typically, if you want to access a persistent object stored in an PSE Pro database, you must open the database and have a session and a transaction in progress, unless you committed (or aborted) your previous transaction using `ObjectStore.RETAIN_READONLY` or `ObjectStore.RETAIN_UPDATE`.

An external reference is an `ExternalReference` object that represents a reference to a persistent object stored in a PSE Pro database.

External references allow you to refer to a persistent object outside a transaction or a session, without opening the database. Even if you cannot access the contents of objects, it is often useful to refer to persistent objects and to pass references to those persistent objects.

For example, you must use a string representation of an external reference when your application

- Passes a reference to an object from one session to another
- Stores a reference in an ASCII file
- Transmits a reference over a serial network connection

For information on the way to create string representations for external references, see Encoding External References as Strings on page 103.

External references can be especially useful when you write a distributed application server that processes requests for many clients. This includes client/server applications that are based on Java RMI (Remote Method Invocation), ObjectStore ObjectForms, or the Object Management Group's Common Object Request Broker Architecture (CORBA).

Be careful of creating external references to objects that might be destroyed or garbage collected before the external reference is used. The garbage collector has no knowledge of an external reference to an object. So when the object being referenced is garbage collected, a tombstone is not left as a placeholder for the object. Therefore, any subsequent dereferencing of the external reference can have unpredictable results. To avoid this situation, Technical Support recommends that you export the referenced object by using `ObjectStore.export()`.

PSE Pro provides the `ExternalReference` class to represent external references. To help you use external references, this section discusses

- Creating External References
- Obtaining Objects from External References
- Encoding External References as Strings
- Using the ExternalReference Field Accessor Methods
- External Reference Equality
- Reusing External Reference Objects
- External Reference Examples

Creating External References

When you create an external reference, you are creating an `ExternalReference` object that represents a reference to a persistent object stored in an PSE Pro database. You cannot create an external reference to a transient object except for an external reference to the Java `null` object.

An external reference identifies a referenced object by storing information about the referenced object's database, segment, cluster, and its location in the cluster.

You can create an external reference by using one of the three `ExternalReference` constructors or by parsing a string representation of an external reference.

Using the no-argument constructor

The no-argument constructor creates an `ExternalReference` object that refers to the `null` object. Creating an external reference using this constructor is equivalent to using the one-argument constructor and passing `null` as the argument.

The constructor signature is

```
public ExternalReference()
```

To use the no-argument constructor, it is not necessary to open a database or to have a session or transaction in progress.

Using the one-argument constructor

The one-argument constructor creates an `ExternalReference` object that identifies the object in the argument. The specified object must be persistent or `null`. The constructor signature is

```
public ExternalReference(Object obj)
```

If the specified object is not `null`, the database that contains the object must be open and a session and transaction must be in progress when the one-argument constructor is called.

Using the four-argument constructor

The four-argument constructor creates an `ExternalReference` object that identifies an object in a specific database, segment, cluster, or specific location in the cluster. You usually obtain the arguments for this constructor by extracting the corresponding fields from a previously created `ExternalReference` object.

The constructor signature is

```
public ExternalReference(Database db, int SegID, int ClusID,
     int loc)
```

A session must be in progress, although it is not necessary for the database to be open or for a transaction to be in progress when the four-argument constructor is called.

Creating an external reference from a string

The `ExternalReference.fromString()` method creates an external reference by parsing and then reconstructing the string it receives from the `ExternalReference.toString()` method.

The method signature is

```
public static ExternalReference fromString(String string)
```

A session must be in progress, although it is not necessary for the database containing the object to be open or for a transaction to be in progress when the `fromString()` method is called.

Obtaining Objects from External References

To obtain the object to which an external reference refers, use the `ExternalReference.getObject()` method.

The method signature is

```
public Object getObject()
```

The database containing the `ExternalReference` object must be open and there must be a session and a transaction in progress when this method is called. The session must be the same session in which the external reference was created.

Note

An external reference is valid only during the session in which it was created. To refer to a persistent object outside a session, you must encode the contents of the external reference in a format that is session neutral, such as a string, or explicitly extract the database name, segment number, cluster number, and location fields from the `ExternalReference` object.

Encoding External References as Strings

Using the `ExternalReference.toString()` and `ExternalReference.fromString()` methods, you can create an external reference that can be used outside the session in which it was created. When the methods are used together, they act as a printer and parser, respectively. They allow you to encode an external reference as a `String`, then parse the string to rebuild an equivalent external reference.

The method signature for encoding an `ExternalReference` object as a `String` is

```
public String toString()
```

The database referred to by the `ExternalReference` object need not be open and a session or a transaction need not be in progress when the `toString()` method is called.

You can pass a string encoding of an external reference to another session or process or store it in an ASCII file.

To reconstruct an `ExternalReference` object from an encoded `String`, call the `fromString()` method. The method signature is

```
public static ExternalReference fromString(String ref)
```

A session must be in progress, although the database referred to by the external reference need not be open and a transaction need not be in progress when the `fromString()` method is called.

The `toString()` and `fromString()` methods are convenient to use, but they create strings that are relatively long because the strings contain the entire pathname of the database. Sometimes you can create a more compact string using accessor methods to extract and then reconstruct an equivalent external reference using the values from the fields in an `ExternalReference` object.

Using the ExternalReference Field Accessor Methods

By extracting the necessary fields from an `ExternalReference` object, you can create a more compact string representation of an external reference that you can use outside a database, process, or session.

Get-accessor methods

To extract the fields of an `ExternalReference` object, you can use the following get-accessor methods:

- public Database getDatabase()
- public int getSegmentId()
- public int getClusterId()
- public int getLocation()

You can store the values of these fields and use them later to reconstruct the external reference.

Note that the `Database` object returned by the `getDatabase()` method is valid only during the session that created the external reference. To encode the `Database` object so that you can use it outside a session, you must translate the database into another form such as a pathname by calling the `Database.getPath()` method. This is exactly what the `ExternalReference.toString()` method does.

Large numbers of external references

If you have a large number of external references for which the database is always the same (or for which the database is known), you might be able to represent the external reference more compactly than you can using the `ExternalReference.toString()` method.

For example, in the case when the database is known, you might be able to represent the external reference by storing just three integers representing the segment identifier, cluster identifier, and the location in the cluster. After you have extracted the database, segment, cluster, and location information, you can use it to reconstruct an external reference.

To reconstruct an external reference, you can use the four-argument constructor or call the set-accessor methods for the corresponding fields:

Set-Accessor methods

- public void setDatabase(Database *d*);
- public void setSegmentId(int *segId*);
- public void setClusterId(int *clustId*));
- public void setLocation(int *loc*);

For example, you can create an external reference that refers to the `null` object using the no-argument constructor, then use the set methods to make the reconstructed external reference refer to a specific object. You can use the set methods repeatedly to update an `ExternalReference` object so that it refers to a variety of objects.

External Reference Equality

Two external references are considered to be equal if they both refer to the same object. In other words, if you call the `ExternalReference.getObject()` method on each external reference, both calls return identical objects.

You call the `ExternalReference.equals()` method to determine whether two external references refer to the same object.

The method signature is

```
public boolean equals(Object obj)
```

Reusing External Reference Objects

After you create an `ExternalReference` object, you can reuse it any number of times. By reusing `ExternalReference` objects, you avoid the overhead of storage allocation and garbage collection when you use large numbers of external references.

You can modify an external reference so that it refers to a different object by using the set-accessor methods or the `setObject()` method.

The method signature is

```
public void setObject(Object obj)
```

External Reference Examples

In the following code fragments, an external reference to `myObj` is created three ways:

- As an external reference
- As an encoded `String`
- By extracting the field values from an external reference, then reconstructing it

First, an external reference (`ref`) is created to `myObj` using the one-argument constructor. This external reference remains valid across transaction boundaries even if the database is closed and reopened. However, the external reference can be used only during the same session in which it was created.

Next, the external reference (`ref`) is encoded as a `String` so that it can be used across session boundaries, then field values are extracted from the external reference to create a more compact representation of the external reference.

Finally, the external reference is resolved to obtain `myObj` using the `getObject()` method for the three types of external references.

```
// Creating an ExternalReference
ExternalReference ref = new ExternalReference(myObj);

// Encoding the ExternalReference as a string
String encodedStr = ref.toString();

// Extracting the fields from the ExternalReference object
Database refDb = ref.getDatabase();
String dbPath = refDb.getPath();
int segId = ref.getSegmentId();
int clustId = ref.getClusterId();
int loc = ref.getLocation();

//...other code...

// If the same session is still active, you can obtain the
// object from the original ExternalReference object:
Object obj1 = ref.getObject();

// If the session is no longer active, you can use the
// encoded string to obtain the object:
ExternalReference ref2 =
  ExternalReference.fromString(encodedStr);
Object obj2 = ref2.getObject();

// Or, you can use the field accessors methods
// to obtain the object:
ExternalReference ref3 = new ExternalReference();
ref3.setDatabase(Database.open(dbPath,
   ObjectStore.READONLY));
ref3.setSegmentId(segId);
ref3.setClusterId(clustId);
ref3.setLocation(loc);
Object obj3 = ref3.getObject();
```

Updating Objects in the Database

To update objects in the database, start at a database root and traverse objects to locate the objects you want to modify. Make your modifications by updating fields or invoking methods on the object, just as you would operate on a transient object. Finally, save your changes by committing the transaction (which ends the transaction), or by evicting the modified objects (which allows the transaction to remain active so the changes can be rolled back by aborting the transaction).

Whether you commit a transaction or evict an object, you can specify the state of objects after the operation. To specify the state that makes the most sense for your application, an understanding of the following background information is important:

- Background for Specifying Object State
- About Object Identity
- About the Object Table

Instructions for invoking `commit()`, and `evict()` follow this background information.

Background for Specifying Object State

When a Java program accesses an object in a PSE Pro database, there are two copies of the object:

- The copy of the object in the database. This is the copy on the disk. It can be anything that is not a primitive. It can be a wrapper object.
- The copy of the object in your Java program. This is the copy that is referred to as a persistent object.

Normally, you need not be aware of the fact that there are two copies. Your application operates on the object in the Java program as if that is the only copy. This is the reason the documentation refers to this copy as a persistent object. However, the fact that there are two copies becomes apparent if a transaction aborts. In this case, the contents of the object in the database revert to the last committed copy. The effect of abort on the copy that is in your Java program depends on the retain mode you used for the abort.

About Object Identity

In a session, persistent objects maintain identity. Suppose there is an object in the database that is referred to by two different objects. You can reach the object in the database through two navigation paths. Regardless of the path you use, the resulting persistent object is the same object in the Java VM. In other words, if you have two unrelated objects (`a` and `b`), that refer to a third object (`c`), `a.c == b.c` is true.

In a single session, the Java VM never creates two distinct objects that both represent the same object in the database.

Sample class definitions

For example, suppose you have the following classes:

```
public class City {
  String name;
  int population;
}

public class State {
  City capital;
  String name;
  int population;
}
```

Creating objects

Suppose you also have the following code, which creates instances of these classes and stores them:

```
City boston = new City("Boston", 1000000);
State massachusetts = new State(
  boston, "Massachusetts", 20000000);
OSHashtable cities = new OSHashtable();
cities.put("Boston", boston);
OSHashtable states = new OSHashtable();
states.put("Massachusetts", massachusetts);
db.createRoot("cities", cities);
db.createRoot("states", states);
```

This creates

- A `City` instance (Boston)
- A `State` instance (Massachusetts) with the Boston `City` instance as its capital
- Two instances of `OSHashtable` — one to hold `City` objects and one to hold `State` objects
- Two database roots — one to refer to each instance of `OSHashtable`

Accessing stored objects

Now you execute the following code to access the stored objects:

```
OSHashtable cities = (OSHashtable) db.getRoot("cities");
OSHashtable states = (OSHashtable) db.getRoot("states");
City boston1 = (City)cities.get("Boston");
State massachusetts = (State)states.get("Massachusetts");
City boston2 = massachusetts.capital;
if (boston1 == boston2)
  System.out.println("same");
else
  System.out.println("not the same");
```

Results

This code prints `"same"`. This is because `boston1` and `boston2`, even though they are located through different paths in the database, are still represented by the same object in the Java VM and, therefore, they are ==.

If you use `cities` to reach `boston1` and you modify `boston1`, you can then use `states` to access the updated version as `boston2`.

Strings and primitive wrappers

There are additional considerations for Strings and primitive wrapper classes.

String pooling causes some strings to be the same object even when you create them separately. If you call `new` multiple times to create multiple `String` objects, these separately created objects might actually refer to the same object when they are

retrieved later in another transaction. See Description of com.odi.stringPoolSize on page 54. If you explicitly migrate the string to the database, it prevents PSE Pro from using string pooling.

A `String` or primitive wrapper object that you create with a single call to `new` might be represented by more than one persistent object when you access it through different paths in subsequent transactions. This happens because a `String` or primitive wrapper object might be stored in the database without the overhead of a regular object. Usually, this does not matter for `String`s and primitive wrapper objects because it is their value and not their identity that matters. If identity does matter, you can explicitly migrate wrapper objects into the database.

Identity across transactions

PSE Pro maintains the identity of referenced objects across transactions within the same session. The following code fragment, displaying `"same"`, provides an example of this:

```
public
class Person {
  // Fields in the Person class:
  String name;
  int age;
  Person children[];
  Person father;
  // Constructor:
  public Person(String name, int age,
    Person children[], Person father) {
    this.name = name;
    this.age = age;
    this.children = children;
    this.father = father;
  }
  static void testIdentity() {
  // Omit open database calls

    Transaction tr = Transaction.begin(ObjectStore.UPDATE);
    Person children[] = { null, null };
    Person tim = new Person("Tim", 35, children, null);
    Person sophie = new Person("Sophie", 5, null, tim);
    children[0] = sophie;
    db.createRoot("Tim", tim);
    tr.commit();

     tr = Transaction.begin(ObjectStore.UPDATE);
    tim = (Person)db.getRoot("Tim");
    Person joseph = new Person("Joseph", 1, null, tim);
    tim.children[1] = joseph;
    tr.commit();

    tr = Transaction.begin(ObjectStore.READONLY);
    tim = (Person)db.getRoot("Tim");
    sophie = tim.children[0];
    joseph = tim.children[1];
    if (sophie.father == joseph.father)
      System.out.println("same");
    else
       System.out.println("not the same");
    tr.commit();
  }
```

About the Object Table

PSE Pro keeps a table of all objects referenced in a transaction. If you refer to the same object in the database twice (perhaps accessing the object through different paths), PSE Pro guarantees that there is only one copy of the object in your Java program. If you retrieve the same object through different paths, `==` returns `true` because PSE Pro preserves object identity.

If the system property `com.odi.disableWeakReferences` is set to `false` (the default), the references in the object table are *weak references*, which means that they do not interfere with the Java GC. If a Java program does not have any references to a persistent object (the copy in your Java program), other than through the PSE Pro object table, the object can be garbage collected. (The object in the database, of course, is not garbage collected.)

Committing Transactions to Save Modifications

When you commit a transaction, PSE Pro

- Saves and commits any modifications in the database
- Checks for transient objects that are referred to by persistent objects

  If there are such objects, PSE Pro stores them in the database if they are persistence capable. This is called transitive persistence. All reachable persistence-capable objects become persistent through transitive persistence.

  If the modifications contain references to objects that are not persistence capable, PSE Pro signals `AbortException`. The `AbortException.getOriginalException()` method returns the object that causes the exception.

- Sets the state of persistent objects after the transaction.

  If objects were stored in the database for the first time during this transaction, the copies of these objects in your Java program are included in the group of persistent objects.

  By default, persistent objects are stale after the transaction. If you do not want to make them stale, there are three ways to specify a default retain state for persistent objects after a commit operation:

- Call `Transaction.commit()` to specify a retain state that applies only to the transaction in which it is called.
- Call `Transaction.setDefaultCommitRetain()` to specify a default retain state that applies to the session in which it is called.
- Call `Transaction.setDefaultRetain()` to specify a default retain state that applies to the session in which it is called.

Method signatures

The `commit()` method has two overloadings. The first overloading takes no argument. The method signature is

```
public void commit()
```

The second overloading has an argument that specifies the state of persistent objects after the commit operation. The method signature is

```
public void commit(int retain)
```

The retain state only applies to the transaction in which the commit was called. You can specify the following retain states after a commit:

- `ObjectStore.RETAIN_HOLLOW`
- `ObjectStore.RETAIN_READONLY`
- `ObjectStore.RETAIN_STALE`
- `ObjectStore.RETAIN_TRANSIENT`
- `ObjectStore.RETAIN_UPDATE`

The values for the *`retain`* argument are described in the sections that follow.

Contents

The following topics in this section describe the different retain states for persistent objects after a commit operation:

- Setting a Default Commit Retain State for a Session
- Setting Persistent Objects to a Default State
- Making Persistent Objects Stale
- Making Persistent Objects Hollow
- Retaining Persistent Objects as Readable
- Retaining Persistent Objects as Writable
- Retaining Persistent Objects as Transient
- The Way Transient Fields Are Handled

Synchronization

If your application needs to synchronize on a persistent object, you might want to retain a reference to that object after a transaction ends by using one of the following ObjectStore constants: `RETAIN_UPDATE`, `RETAIN_HOLLOW`, or `RETAIN_READONLY`. When persistent objects become stale, PSE Pro does not maintain their transient identity. Their synchronized states are not saved persistently.

Setting a Default Commit Retain State for a Session

You can use the following two methods to set the default retain state for persistent objects after a transaction is committed:

- `Transaction.setDefaultCommitRetain()`
- `Transaction.setDefaultRetain()`

The default retain state set by these methods is in effect for the duration of the session in which they are called.

Use `setDefaultCommitRetain()` to specify a default retain state for persistent objects when `Transaction.commit()` is called without a `retain` argument. The `setDefaultCommitRetain()` method also sets the default retain state for persistent objects when `Transaction.checkpoint()` is called without a `retain` argument. The method signature is

```
public void setDefaultCommitRetain(int retain)
```

The values you can specify for *`retain`* are the same values you can specify when you call `commit()` with a *`retain`* argument. These values are described in the sections that follow.

Use `setDefaultRetain()` to specify a default commit retain state for persistent objects when `Transaction.commit()` is called without a *`retain`* argument. This method also sets the default retain states for persistent objects when `Transaction.abort()` and `Transaction.checkpoint()` are called without a *`retain`* argument. The method signature is

```
public void setDefaultRetain(int retain)
```

Note

When you are using either method to set a default retain state, the method that was last called overrides any default retain state that was set previously. For example, if an application calls `setDefaultCommitRetain()` first and then calls

`setDefaultRetain()`, the retain state for `checkpoint()` and `commit()` is specified by the second call.

Setting Persistent Objects to a Default State

To commit a transaction and to set the state of persistent objects to the state specified by `Transaction.setDefaultCommitRetain()` or by `Transaction.setDefaultRetain()`, call the `commit()` method without any arguments. For example:

```
tr.commit();
```

Making Persistent Objects Stale

When you call the `commit()` method with no argument, PSE Pro makes all persistent objects stale unless a default retain state has been specified previously by either the `setDefaultCommitRetain()` or `setDefaultRetain()` method. Stale persistent objects are not accessible and their contents are set to default values. PSE Pro reclaims the entry in the Object Table for the stale object and the object loses its persistent identity.

If your Java program still has references to stale objects, any attempt to use those references, such as by accessing a field or calling a method on the object, causes PSE Pro to signal `ObjectException`. Therefore, your application must discard any references to persistent objects when it calls this overloading of `commit()`.

Objects available for garbage collection

This overloading of `commit()` also discards any internal PSE Pro references to the copies of the objects in your Java program. When your application makes an object stale, PSE Pro makes any references from the stale object to other objects null. This makes the referenced objects, which can be persistent or transient, available for garbage collection if there are no other references to them from other objects.

Stale persistent objects are not available for Java garbage collection if your Java application has transient references to them.

Accessing objects again

You can reaccess the same objects in the database in subsequent transactions. To do so, look up a database root and traverse to objects from there, or reference them through hollow objects. PSE Pro refetches the contents of the object and creates a new active persistent object. The new object has a new transient identity and the same persistent identity as the object that became stale.

For example:

```
Foo foo = myDB.getRoot("A_FOO");
ExternalReference fooRef = new ExternalReference(foo);
ObjectStore.evict(foo, ObjectStore.RETAIN_STALE);
Foo fooTwo = myDB.getRoot("A_FOO");  // refetch from database
ExternalReference fooRefTwo = new ExternalReference(fooTwo);
// At this point (foo == fooTwo) returns false,
// but (fooRef.equals(fooTwoRef)) returns true.
```

Advantage

The advantage of using `commit()` with no argument when a default retain type is *not* specified, is that it wipes your database cache clean and typically makes all transient copies of persistent data available for Java garbage collection.

Disadvantage

The disadvantage of using `commit()` is that any references to these objects that your Java program holds become unusable unless a default retain state was previously specified.

Alternative method

Invoking `commit(ObjectStore.RETAIN_STALE)` is the same as calling `commit()` with no argument unless a default retain state was previously specified.

Making Persistent Objects Hollow

Call `commit(ObjectStore.RETAIN_HOLLOW)` to make persistent objects (the copies of the objects in your Java program) hollow. PSE Pro resets the contents of persistent objects to default values.

References to these objects remain valid; the application can use them in a subsequent transaction. If a hollow object is accessed in a subsequent transaction, PSE Pro refreshes the contents of the object in your Java program with the contents of the corresponding object in the database.

Outside transaction

An application cannot access hollow objects outside a transaction. An attempt to do so causes PSE Pro to signal `NoTransactionInProgressException`.

Advantage

The advantage of invoking `commit(ObjectStore.RETAIN_HOLLOW)` is that any references to persistent objects that the Java application holds remain valid in subsequent transactions. This means that it is not necessary to renavigate to these objects from a database root.

Disadvantage

The disadvantage of retaining persistent objects as hollow objects is that in Java VM implementations for which PSE Pro does not have weak reference support, hollow persistent objects are not available for Java garbage collection. This is true regardless of whether or not your Java program has references to these objects.

Garbage collection

Sometimes an application might retain a reference to an object and prevent Java garbage collection that would otherwise occur. It is good practice to avoid retaining references to objects unnecessarily.

Scope

If you commit a transaction with `ObjectStore.RETAIN_HOLLOW`, then commit a subsequent transaction with no `retain` argument or `ObjectStore.RETAIN_STALE`, this cancels the previous `ObjectStore.RETAIN_HOLLOW` specification. No object references are available in the next transaction. This is true regardless of whether they were previously retained.

Retaining Persistent Objects as Readable

Call `commit(ObjectStore.RETAIN_READONLY)` to retain the copies of the objects in your Java program as readable persistent objects. PSE Pro maintains the contents of the persistent objects that the application read in the transaction just committed. The contents of these persistent objects are as they were the last time the objects were read or modified in the transaction just committed.

If any hollow objects exist when you commit the transaction, PSE Pro retains these objects as hollow objects that you can use during the next transaction.

After this transaction and before the next transaction, your application can read the contents of any retained objects whose contents were also retained. The actual contents of the object in the database might be different because another process modified it. Your application cannot modify these objects. An attempt to do so causes PSE Pro to signal `NoTransactionInProgressException`. Your application cannot access the contents of hollow retained objects. An attempt to do so causes PSE Pro to signal `NoTransactionInProgressException`.

Scope

If you commit a transaction with `ObjectStore.RETAIN_READONLY`, the contents of only those persistent objects whose contents were accessed in the transaction just committed are available to you after the transaction. This is because PSE Pro makes all retained objects hollow at the start of the next transaction. Any cached references to persistent objects remain valid. In the new transaction, PSE Pro fetches the contents of a persistent object when your application requires it.

In previous releases, PSE Pro did not make retained objects hollow at the start of a new transaction. If your application needs the old behavior, there is a system property you can set. See Description of com.odi.disableCrossTransactionCaching on page 53.

Advantage

The advantage of using `commit(ObjectStore.RETAIN_READONLY)` is that the copies of the persistent objects in your Java program remain accessible after the transaction is over. In the next transaction, any cached references to persistent objects remain valid. PSE Pro copies the object's contents from the database when you access the object.

Disadvantage

The disadvantage of using `commit(ObjectStore.RETAIN_READONLY)` is that it makes more work for the Java GC, because the contents of the copies of the objects in your Java program are not cleared.

Your program might return results that are inconsistent with the current state of the database.

PSE Pro cannot fetch any objects outside a transaction. This makes it difficult to ensure that methods can execute without signaling an exception. However, you can call `ObjectStore.deepFetch()` in the transaction to obtain the contents of all objects you might need. Of course, this increases the risk of the Java VM's running out of memory.

Serialization

If you are using Java Remote Method Invocation (RMI) or serialization, you can call the `ObjectStore.deepFetch()` method followed by `commit(ObjectStore.RETAIN_READONLY)`. This allows you to perform object serialization outside a transaction.

Between transactions, you might try to read an object that you thought you retained, and receive `NoTransactionInProgressException`. Often, the cause of this is that you retained a reference to the object but not the contents of the object.

Trouble-shooting

In a transaction, you might read the contents of an object but not the contents of objects the first object refers to. For example, during a transaction, suppose you access a vector but not any of the elements in the vector. When you commit the transaction, the contents of vector elements are not available in the transaction and they are not retained. In other words, to be able to read the contents of an object between transactions, you must read that particular object during the previous transaction.

To be able to read objects between transactions, you might want to call `ObjectStore.deepFetch()` on an object. This method fetches the contents of the specified object, the contents of any objects that object refers to, the contents of any objects those objects refer to, and so on for all reachable objects.

Inside a transaction, PSE Pro fetches the contents of objects automatically as you read the objects. Outside a transaction, if a reference to an object, but not the contents of the object, was retained, PSE Pro signals `NoTransactionInProgressException`.

Following is another situation in which you would receive the `NoTransactionInProgressException`:

1. In a transaction, you read object `A`.
2. You commit the transaction with `ObjectStore.RETAIN_READONLY`.
3. You start a new transaction. It does not matter whether you access object `A` in this transaction.
4. You commit this transaction with `ObjectStore.RETAIN_STALE` or without a `retain` argument.
5. Outside a transaction, you try to access object `A` and you receive the `NoTransactionInProgressException`.

You might think that because you retained `A` after a previous transaction, its contents are still available. This is not the case. Because nothing was retained after the second transaction, the contents of `A` are no longer available.

Retaining Persistent Objects as Writable

Call `commit(ObjectStore.RETAIN_UPDATE)` to retain the copies of the objects in your Java program as readable and writable. PSE Pro maintains the contents of the persistent objects as they are at the end of the transaction.

Sometimes, the contents of an object are not available in a transaction when you expect that they are available, such as when you receive a `NoTransactionInProgressException`. For more information about what causes this exception, see Retaining Persistent Objects as Readable on page 114.

Specifying `ObjectStore.RETAIN_UPDATE` is exactly like specifying `ObjectStore.RETAIN_READONLY`, except that if your application accesses the objects after the transaction and before the next transaction, your application can modify as well as read the objects. At the beginning of the next transaction, PSE Pro discards any updates made to the persistent objects between transactions.

Retaining Persistent Objects as Transient

Call `commit(ObjectStore.RETAIN_TRANSIENT)` to convert all persistent objects associated with a session into transient objects at the end of a transaction. Because these transient objects are copies of persistent objects, they no longer represent persistent objects in a database. As a result, these transient objects can be read or modified outside transactions because they are no longer associated with a database.

Unlike objects retained at the end of transactions by using the `RETAIN_READONLY` or `RETAIN_UPDATE` arguments, objects retained by using the `RETAIN_TRANSIENT` argument no longer represent persistent objects in a database. To reacquire the corresponding persistent object that the transient object (copy) once represented, the persistent object must be fetched again from the database. As a result of refetching the persistent object, you now have two Java objects in the Java VM. One Java object is the copy of the persistent object that is no longer associated with the database; the other Java object (the one just fetched) represents the persistent object in the database. If you perform an object identity test on these two transient objects (`O1 == O2`), the test returns `false` because they are not the same object.

You should be aware that after committing a transaction by using the `RETAIN_TRANSIENT` argument, PSE Pro does not prevent your application from accessing a hollow object that is referenced from a Java object that was retained as transient. The fields of this hollow object will contain values such as `0` and `null`.

Garbage collection

Just like other Java objects, objects that are retained at the end of a transaction by using the `RETAIN_TRANSIENT` argument are candidates for garbage collection when they are no longer referenced by other objects.

Advantage

The `RETAIN_TRANSIENT` argument is a faster alternative to `Database.close(retainAsTransient)` because the `RETAIN_TRANSIENT` argument does not require the database to be opened and closed, which involves deallocating and reallocating all the resources associated with the database.

Using the `RETAIN_TRANSIENT` argument is a way to copy data from a database into the Java VM. Once the data is in the Java VM, you can manipulate the data without throwing database exceptions.

Disadvantage

You must code your program carefully so that it does not inadvertently manipulate a copy of a persistent object instead of the actual transient object representing the persistent object stored in the database.

The Way Transient Fields Are Handled

PSE Pro does not modify the contents of any transient-only fields unless you have explicitly defined one of the following methods to modify transient-only fields:

- `IPersistent.clearContents()`
- `IPersistent.preClearContents()`
- `IPersistent.initializeContents()`
- `IPersistent.postInitializeContents()`

The `clearContents()` and `initializeContents()` methods that are generated by the postprocessor do not modify transient-only fields.

Advantage

The advantage of using `commit(ObjectStore.RETAIN_UPDATE)` is that the copies of the objects in your Java program become scratch space between transactions. You can use them to determine the possible results for a particular scenario.

Disadvantage

The disadvantage is that updates are discarded automatically at the beginning of the next transaction. This can make it difficult to debug applications that use this option indiscriminately.

Evicting Objects to Save Modifications

You might want to save modifications to an object or change the state of an object without committing a transaction. The `evict()` method allows you to do this.

Method signatures

The `evict()` method has two overloadings. The first overloading takes an object as an argument. The method signature is

```
public static void evict(Object object)
```

The second overloading has an additional argument that specifies the state of the evicted object after the eviction. The method signature is

```
public static void evict(Object object, int retain)
```

This section provides the following information about evicting objects:

- Description of Eviction Operation
- Setting the Evicted Object to Be Stale
- Setting the Evicted Object to Be Hollow
- Setting the Evicted Object to Be Read-Only
- Summary of Eviction Results for Various Object States
- Evicting All Persistent Objects
- Evicting Objects When There Are Cooperating Threads
- Committing Transactions After Evicting Objects
- Evicting Objects Outside a Transaction

Description of Eviction Operation

When you evict an object, PSE Pro

- Saves any modifications to the object in the database but does not commit the changes. If the transaction commits, any changes are committed. If the transaction aborts, the contents of the object in the database revert to the contents following the last committed transaction in which the object was modified.
- Sets the state of the evicted object after the eviction. This affects the copy of the object that is in your Java program. The default is that the evicted object is stale after the eviction. If you do not want it to be stale, you can specify another state when you invoke `evict()`.

References to other objects

When you evict an object, PSE Pro does not evict objects that the evicted object references.

You might evict an object that has instance variables that are transient strings. PSE Pro migrates such strings to the database and stores them in the same segment as the evicted object. As part of the eviction process, PSE Pro evicts the just-stored string with a specification of `ObjectStore.RETAIN_READONLY`. Consequently, after the eviction, the migrated string remains readable.

Caution

You must ensure that a persistent object never refers to a persistent object that belongs to a different session. PSE Pro throws an exception if it reaches a persistent object belonging to one session while it is performing transitive persistence for another session.

Setting the Evicted Object to Be Stale

When you invoke `evict(Object)`, PSE Pro makes the evicted object stale. PSE Pro resets the contents of the copy of the object in your Java program to default values and makes the object inaccessible. Any references to the evicted object are stale and your application should discard them. The copy of the object in your Java program becomes available for Java garbage collection.

Advantage

The advantage of using the `evict(Object)` method is that the evicted object and all objects it refers to become available for Java garbage collection (if they are not referenced by other objects in the Java program).

Disadvantage

The disadvantage is that any references to the evicted object become stale. If you try to use the stale references, PSE Pro signals `ObjectException`.

The effect on accessibility to the copy of the object in your Java program is therefore similar to the effect of `commit()` and `commit(ObjectStore.RETAIN_STALE)`. However, if the transaction aborts, any changes to the evicted object are discarded.

Alternative method

A call to `evict(Object, ObjectStore.RETAIN_STALE)` is identical to a call to `evict(Object)`.

Setting the Evicted Object to Be Hollow

When you invoke `evict(Object, ObjectStore.RETAIN_HOLLOW)`, PSE Pro makes the evicted object hollow. PSE Pro resets the contents of the copy of the object in your Java program to default values. References to the evicted object continue to be valid.

If the application accesses the evicted object in the same transaction, PSE Pro copies the contents of the object from the database to the copy in your Java program. If your application modified the object before evicting it, these modifications are included in the new copy in your Java program.

Advantage

The reason to use the `evict(Object, ObjectStore.RETAIN_HOLLOW)` method is that the object and all objects it refers to become available for Java garbage collection (if they are not referenced by other objects in the Java program).

Disadvantage

The reason not to use this method is that in Java VM implementations for which PSE Pro does not have weak reference support, hollow persistent objects are not available for Java garbage collection.

Sometimes an application might retain references to an object and prevent Java garbage collection that would otherwise occur. It is good practice to avoid retaining references to objects unnecessarily.

Setting the Evicted Object to Be Read-Only

When you invoke `evict(Object, ObjectStore.RETAIN_READONLY)`, PSE Pro

- Retains references to the evicted object.
- Retains the contents of the evicted object.
- Saves any changes to the evicted object.
- Internally flags the object as read but not modified. This is because any changes are already saved. If the application later modifies the evicted object in the same transaction, PSE Pro modifies this flag accordingly.

Garbage collection

Any changes that were made before the object was evicted are saved in the database. (Of course, if the transaction aborts, the changes are rolled back.) Therefore, if the evicted object is not referenced by other objects in the Java program, it becomes available for Java garbage collection.

Additional changes

Your application can read or modify the evicted object in the same transaction. If it does, PSE Pro does not have to recopy the contents of the object from the database to your program. When the application commits the transaction or evicts the object again, PSE Pro saves in the database any new changes to the evicted object.

It might seem strange to evict an object with `ObjectStore.RETAIN_READONLY` and yet be able to modify the object after the eviction. The specification of `READONLY` in this context means that as of this point in time, the evicted object has been read but not modified. The changes have already been saved but not committed. The contents are still available and can be read or updated.

Advantage

The advantage of using `evict(Object, ObjectStore.RETAIN_READONLY)` is that the updated object becomes available for Java garbage collection on platforms that support weak references.

Summary of Eviction Results for Various Object States

The following table shows the results of an eviction according to the value specified for the *`retain`* argument.

Results of Eviction	`RETAIN_STALE`	`RETAIN_HOLLOW`	`RETAIN_READONLY`
Object state	Stale	Hollow	Active
References to evicted object	Stale	Remain valid	Remain valid
Candidate for Java garbage collection	Candidate	Can be candidate	Can be candidate

Evicting All Persistent Objects

You can evict all persistent objects associated with the current session with one call to `evictAll()`. The method signature is

```
public static void evictAll(int retain)
```

For the *`retain`* argument, you can specify

- `ObjectStore.RETAIN_HOLLOW`
- `ObjectStore.RETAIN_READONLY`
- `ObjectStore.RETAIN_STALE`
- `ObjectStore.RETAIN_TRANSIENT`

When you specify any of the *`retain`* arguments, PSE Pro applies it to all persistent objects that belong to the same session as the active thread. PSE Pro does this in the same way that it applies a `retain` argument to one object for the `evict(object, retain)` method. Note that `evict(object, retain)` does not have `RETAIN_TRANSIENT` as a value for the *`retain`* argument.

`ObjectStore.RETAIN_TRANSIENT` converts all persistent objects associated with a session into transient objects after evicting all objects from the transaction. Because these transient objects are copies of persistent objects, they no longer represent persistent objects in a database. You can read or modify the objects outside transactions.

Evicting Objects When There Are Cooperating Threads

Before an application evicts an object, it must ensure that no other thread requires that object to be accessible. For example, suppose you have code like the following:

```
class C {
  String x;
  String y;

  void function() {
    System.out.println(x);
    ObjectStore.evict(this);
    System.out.println(y);
  }
}
```

Before the first call to `println()`, the object is accessible. After the call to `evict()`, the `y` field is null and the second `println()` call fails. There are more complicated scenarios for this problem that involve subroutines that call `evict()` and cause problems in the calling functions. This problem can occur in a single thread. If there are multiple cooperating threads, each thread must recognize what the other thread is doing. See Cooperating Threads on page 47.

It is the responsibility of the application to ensure that the object being evicted is not the `this` argument of any method that is currently executing.

Committing Transactions After Evicting Objects

In a transaction, you might evict certain objects and specify their states to be hollow or active. If you then commit the transaction and cause the state of persistent objects to be stale, this overrides the hollow or active state set by the eviction. If you commit the transaction and cause the state of persistent objects to be hollow, this overrides an active state set by eviction. For example:

```
Transaction tr = Transaction.begin(ObjectStore.UPDATE);
Trail trail = (Trail) db.getRoot("greenleaf");
GuideBook guideBook = trail.getDescription();
ObjectStore.evict(guideBook, ObjectStore.RETAIN_READONLY);
tr.commit();
```

After the transaction commits, the application cannot use `guideBook`. Committing the transaction without specifying a `retain` argument makes all persistent objects stale (unless a `retain` value other than `RETAIN_STALE` was specified by `Transaction.setDefaultCommitRetain()` or `Transaction.setDefaultRetain()`). This overrides the `RETAIN_READONLY` specification when `guideBook` was evicted.

Evicting Objects Outside a Transaction

Outside a transaction, eviction of an object has meaning only if you retained objects when you committed the previous transaction. In other words, if you invoke the `commit(retain)` method and specify a value for the `retain` argument other than `RETAIN_STALE`, you can evict retained objects outside a transaction.

If you specified `commit(ObjectStore.RETAIN_STALE)`, there are no objects to evict after the transaction commits. If you invoked `commit()` with any other retain value, you can call `evict()` or `evictAll()` with the value of the `retain` argument as `RETAIN_STALE` or `RETAIN_HOLLOW`. If you specify `RETAIN_READONLY`, PSE Pro does nothing.

Outside a transaction, if you make any changes to the objects you evict, PSE Pro discards these changes at the start of the next transaction. They are not saved in the database.

Aborting Transactions to Cancel Changes

If you modify some objects, then decide that you do not want to keep the changes, you can abort the transaction. Aborting a transaction

- Ensures that the objects in the database are as they were just before the aborted transaction started
- Sets the state of persistent objects from the transaction

Only the state of the database is rolled back. The state of transient objects is not undone automatically. Applications are responsible for undoing the state of transient objects. Any form of output that occurred before the abort cannot be undone.

Method signatures

The `abort()` method has two overloadings. The first overloading takes no argument. The method signature is

```
public void abort()
```

The second overloading has an argument that specifies the state of persistent objects after the abort operation. The method signature is

```
public void abort(int retain)
```

The retain state only applies to the transaction in which the abort was called. You can specify the following retain states after a abort:

- `ObjectStore.RETAIN_HOLLOW`
- `ObjectStore.RETAIN_READONLY`
- `ObjectStore.RETAIN_STALE`
- `ObjectStore.RETAIN_UPDATE`
- `ObjectStore.RETAIN_TRANSIENT`

The values for the *`retain`* argument are described in the sections that follow.

Contents

This section discusses the following topics:

- Setting a Default Abort Retain State for a Session
- Setting Persistent Objects to a Default State
- Specifying a Particular State for Persistent Objects

Setting a Default Abort Retain State for a Session

You can use the following two methods to set the default retain state for persistent objects after a transaction is aborted:

- `Transaction.setDefaultAbortRetain()`
- `Transaction.setDefaultRetain()`

The default retain state set by these methods is in effect for the duration of the session in which they are called.

Call the `setDefaultAbortRetain()` method to set the default state for persistent objects after a transaction is aborted. The default retain state is in effect for the duration of the session in which it is called. The method signature is

```
public void setDefaultAbortRetain(int newRetain)
```

The values you can specify for *`newRetain`* are the same values you can specify when you call `abort()` with a *`retain`* argument. These values are described in the next section. Another way to set the default state for persistent objects after a transaction is aborted is to call the `setDefaultRetain()` method. The method signature is

```
public void setDefaultRetain(int retain)
```

The `setDefaultRetain()` method also sets the default retain states for `Transaction.commit()` and `Transaction.checkpoint()` when these methods are called without a *`retain`* argument. The default retain state for persistent objects is in effect for the duration of the session in which it is called.

Note

When you are using either method to set a default retain state, the method that was last called overrides any default retain state that was set previously. For example, if an application calls `setDefaultAbortRetain(int retain)` first and then calls `setDefaultRetain(int retain)`, the retain state for `abort()` is specified by the second call.

Setting Persistent Objects to a Default State

To abort a transaction and to set the state of persistent objects to the state specified by `Transaction.setDefaultAbortRetain()` or by `Transaction.setDefaultRetain()`, call the `abort()` method without any arguments. The default state is stale if a default retain state is not specified.

The method signature is

```
public void abort()
```

For example:

```
tr.abort();
```

Specifying a Particular State for Persistent Objects

To abort a transaction and to specify a particular state for persistent objects after the transaction, call the `abort(retain)` method on the transaction. The method signature is

```
public void abort(int retain)
```

The *`retain`* value you specify affects the retain state of the persistent objects for the transaction in which it is called.

The following example aborts a transaction and specifies that the contents of the active persistent objects should remain available to be read:

```
tr.abort(ObjectStore.RETAIN_READONLY);
```

The values you can specify for *`retain`* are described next.

The rules for Java garbage collection of objects retained from aborted transactions are the same as for objects retained from committed transactions. See Committing Transactions to Save Modifications on page 111.

RETAIN_STALE

`ObjectStore.RETAIN_STALE` resets the contents of all persistent objects to their default values and makes them stale. This is the same as calling `abort()` when `Transaction.setDefaultAbortRetain()` or `Transaction.setDefaultRetain()` have not been called or one has been called with `ObjectStore.RETAIN_STALE` as its argument.

RETAIN_HOLLOW

`ObjectStore.RETAIN_HOLLOW` resets the contents of all persistent objects to their default values and makes them hollow. In the next transaction, you can use references to persistent objects from this transaction.

RETAIN_READONLY

`ObjectStore.RETAIN_READONLY` retains the contents of unmodified persistent objects that were read during the aborted transaction. Any objects that were modified become hollow objects, as if `ObjectStore.RETAIN_HOLLOW` had been specified. Objects whose contents were read but not modified in the aborted transaction can be read after the aborted transaction.

If you try to modify a persistent object before the next transaction, PSE Pro signals `NoTransactionInProgressException.` If you modified any persistent objects during the aborted transaction, PSE Pro discards these modifications and makes these objects hollow as part of the abort operation.

During the next transaction, the contents of persistent objects that were not modified during the aborted transaction are still available.

RETAIN_UPDATE

`ObjectStore.RETAIN_UPDATE` retains the contents of persistent objects that were read or modified during the aborted transaction. The values that are retained are the last values that the objects contained before the transaction was aborted. Even though the changes to the modified objects are undone with regard to the database, the changes are not undone in the objects in the Java VM.

While you are between transactions, the changes that were aborted are still visible in the Java objects. At the start of the next transaction, PSE Pro discards the modifications and reads in the contents from the database. Objects that were read or modified in the aborted transaction can be modified between the aborted transaction and the next transaction. If you modify any persistent objects during or after the aborted transaction, PSE Pro discards these modifications and makes these object hollow at the start of the next transaction.

During the next transaction, the contents of persistent objects that were not modified during or after the aborted transaction are still available.

RETAIN_TRANSIENT

`ObjectStore.RETAIN_TRANSIENT` converts all persistent objects associated with a session into transient objects after aborting a transaction. Because these transient objects are copies of persistent objects, they no longer represent persistent objects in a database. They can be read or modified outside transactions.

Destroying Objects in the Database

You can explicitly destroy any object that you want to be deleted from persistent storage. The discussion of the destroy operation covers the following topics:

- Calling ObjectStore.destroy()
- Destroying Objects That Refer to Other Objects
- Destroying Objects That Are Referred to by Other Objects
- Effects of Destroying an Object on page 129

Calling ObjectStore.destroy()

To destroy an object, call `ObjectStore.destroy()`. This method has two overloadings:

```
public static void destroy(Object object)
public static void destroy(Object object, DestroyOptions option)
```

The object you specify must be persistent or the call has no effect. The database that contains the object must be open for update and an update transaction must be in progress.

If you use the overloading that does not specify the second argument, it is as if you had specified the second argument with a value of `DETECT_DANGLING_REFERENCES`. If you try to access the destroyed object, the tombstone causes PSE Pro to signal `ObjectNotFoundException`.

If you specify the `option` argument, the value must be one of the values for the `DestroyOptions` enum:

- `ASSERT_NO_DANGLING_REFERENCES` — When PSE Pro destroys the object, it does not leave a tombstone at the location occupied by the destroyed object.

  CAUTION: Specify this option only if the application is certain that there are no references to the object being destroyed. Database corruption can occur if you specify this option when there are references to the destroyed object in the database.

- `DETECT_DANGLING_REFERENCES` — When PSE Pro destroys the object, it leaves a tombstone at the location of the destroyed object. If there is a reference to the destroyed object and the application tries to navigate through that reference, PSE Pro signals an exception.

When the `com.odi.forceTombstones` property is set to true, the `destroy()` method always leaves a tombstone even if you specify `ASSERT_NO_DANGLING_REFERENCES` when you call the `destroy()` method.

If the destroyed object either implements the `IPersistent` interface or is an array, you cannot access any of its fields after you destroy it.

Destroying Objects That Refer to Other Objects

By default, when you destroy an object, PSE Pro does not destroy objects that the destroyed object references.

There is a hook method, `IPersistentHooks.preDestroyPersistent()`, that you can define. PSE Pro calls this method before actually destroying the specified object. This method is useful when an object has underlying structures that you want to destroy along with the object. The default implementation of this method does nothing.

You can use `preDestroyPersistent()` to propagate the destroy operation to child objects that are referenced by the one being destroyed. If you do this, be careful that the child objects themselves are not referenced by other objects in the database. If an object attempts to use a reference to an explicitly destroyed object, PSE Pro signals

`ObjectNotFoundException`. If you are not certain whether a specific object might be referenced elsewhere, it is better to avoid explicitly destroying the object. Let the persistent GC do the job instead.

OSHashtable and OSVector

When you delete a `com.odi.util.OSHashtable` or `com.odi.util.OSVector` object, PSE Pro deletes the hash table or vector and its own internal data structures. PSE Pro does not delete the keys or elements that were inserted into the hash table or vector. Doing so might cause problems because other Java objects might refer to those objects.

However, sometimes you want to destroy the objects in a hash table or vector as well as the hash table or vector itself. Suppose you have a class in which one of the instance variables is a `com.odi.util.OSVector`. You might want to ensure that whenever an instance of this class is destroyed, the `OSVector` and its contents are also destroyed. To do this, you can define a `preDestroyPersistent()` method on your class. Define this method to iterate over the elements in the vector, destroy each one, then destroy the `com.odi.util.OSVector`.

Types not destroyed

When you call the `ObjectStore.destroy()` method on an object, it does not destroy fields in the object that are

- `String` types
- Instances of wrapper classes that have been explicitly migrated with the `ObjectStore.migrate()` method
- `Long` and `Double` types

For additional information about PSE Pro treatment of `String` instances, see Description of Special Behavior of String Literals on page 265. For example, if you define a class such as the one following, when you destroy an instance of this class, you should also explicitly destroy `s` and `d`.

```
class C {
  int i;
  String s;
  Double d;
}
```

Advantages of explicit destroy

You should always consider whether or not to have `preDestroyPersistent()` call `ObjectStore.destroy()` on fields that contain `String` types, instance of wrapper classes that have been explicitly migrated, or types that you define. The advantage of explicitly destroying objects is that PSE Pro replaces large objects or arrays with a 4-byte tombstone.

Disadvan-tages of explicit destroy

The disadvantages of explicitly destroying such objects are

- You must write additional code.
- There is the risk of a dangling reference if you are not careful. For example, an unanticipated `ObjectException` might prevent an object from being destroyed.
- PSE Pro replaces a destroyed object with a tombstone that uses 4 bytes. This can cause fragmentation. The tombstone can also cause PSE Pro to signal `ObjectNotFoundException`. For example, suppose you unintentionally destroy an object that is referenced by another object. When you try to dereference the reference to the destroyed object, the tombstone causes PSE Pro to signal `ObjectNotFoundException`.

You need not have `preDestroyPersistent()` call `ObjectStore.destroy()` on fields that contain primitive types.

Example

For example, suppose you have a persistence-capable class called `MyVector` that has a private field called `contents`. When an instance of `MyVector` is persistent, the `contents` field is also persistent, but a user would not have access to it because it is private. If a user calls `ObjectStore.destroy()` on an instance of `MyVector`, the operation destroys the instance but not the `contents` object.

If you are the programmer implementing the `MyVector` class, you have two choices:

- Provide a `MyVector.`*`destroy`*`()` method to call `ObjectStore.destroy(contents)`. If you do this, you must ensure that users of `MyVector` understand that they should not call `ObjectStore.destroy()` on an instance of `MyVector` because doing so leaves garbage in the database.
- Provide a `preDestroyPersistent()` method that calls `ObjectStore.destroy(contents)`. This choice ensures that if a user calls `ObjectStore.destroy()` on an instance of `MyVector`, the operation cleans up the private `contents` field.

Following is code that shows the second alternative:

```
public class MyVector {
  private Object[] contents;

  public addElement(Object o) {
    contents[nextElement++] = o;
  }

  public void preDestroyPersistent(DestroyOptions option) {
    if (contents != null)
      ObjectStore.destroy(contents, option);
  }
}
```

Destroying Objects That Are Referred to by Other Objects

The usual practice is to remove references to a persistent object before you destroy that persistent object. PSE Pro signals `ObjectNotFoundException` when you try to access a destroyed object. It is up to you to clean up any references to destroyed objects.

If an object retains a reference to a destroyed object, PSE Pro signals `ObjectNotFoundException` when you try to use that reference. This might occur long after the referenced object was destroyed. To clean up this situation, set the reference in the referring object to null.

String class and wrapper objects

A call to `destroy` on a `String` object or wrapper object behaves differently. When you dereference a reference to such a destroyed object, PSE Pro does not signal `ObjectNotFoundException`. Instead, references to the destroyed object from objects modified in the same transaction as the destroy operation continue to have the value of the destroyed object. References to the destroyed object from objects not modified in the same transaction appear as null values when an object containing such a reference is fetched.

Hash tables

You should avoid having a hash table refer to a destroyed object. It is difficult to remove a reference from a hash table after you destroy the object that it refers to. This is because the search through the hash table for the referring object might cause PSE Pro to try to access the destroyed object. In fact, a search for another object in the hash table might cause PSE Pro to access the destroyed object. The result is that the hash table look-up procedure signals `ObjectNotFoundException` and the hash table becomes useless. Consequently, you should always remove objects from hash tables before you destroy them.

Effects of Destroying an Object

When an application destroys an object, PSE Pro makes the space occupied by that object in the `.odb` and `.odt` files available for reuse by other objects in future transactions. Your application can use the space in the `.odb` file in the next transaction. You can use the space in the `.odt` file in a later transaction. For both files, the free space is reused, but the files do not shrink.

Default Effects of Various Methods on Object State

The following table summarizes the default effects of various methods on the state of hollow or active persistent objects. You should never try to invoke a method on a stale object. If you do, PSE Pro tries to detect it and signal `ObjectException`. PSE Pro can signal `ObjectException` for objects that are instances of classes that implement the `IPersistent` interface.

Unless you manually annotate your classes to make them persistence capable, you do not write `ObjectStore.fetch()` or `ObjectStore.dirty()` calls in your application. The postprocessor inserts these calls automatically as needed.

The information in the following table assumes that you are not specifying a `retain` argument with any of the methods that accept a `retain` argument.

Method the Application Calls	*Result When Invoked on Hollow or Active Objects*
`ObjectStore.fetch()`	Active persistent object
`ObjectStore.dirty()`	Active persistent object
`ObjectStore.evict()`	Hollow persistent object
`ObjectStore.destroy()`	Stale persistent object

Method the Application Calls	*Result*
`Transaction.commit()`	Persistent objects become stale.
`Transaction.abort()`	Persistent objects become stale.

Transient Fields in Persistence-Capable Classes

This section discusses

- Behavior of Transient Fields
- Preventing fetch() and dirty() Calls on Transient Fields

See also Creating Persistence-Capable Classes with Transient Fields on page 196.

Behavior of Transient Fields

In a persistence-capable class, a field designated with the `transient` keyword behaves as follows:

- A transient field is never stored in a database.
- A transient field can be initialized in a constructor just like any other field.
- When an object is materialized from a database, a transient field has the value that the constructor gives it.
- By overriding the `postInitializeContents()` method, you can synchronize a transient field for an object when its contents are refreshed from the database.
- When an object becomes hollow or stale, a transient field is not cleared.
- If you assign the value of a persistent object to a transient field, all memory of the reference is lost when the enclosing object is garbage collected.
- If you try to access a transient field outside a transaction, PSE Pro signals `NoTransactionInProgressException` if the containing object is hollow or `ObjectException` if the containing object is stale.
- Committing or aborting a transaction has no effect on a transient field.

Preventing fetch() and dirty() Calls on Transient Fields

When you run the postprocessor on a class that has transient fields, you might want to specify the `-noannotatefield` option for the transient fields. This option prevents access to the specified field from causing `fetch()` and `dirty()` calls on the containing object. This is useful for transient fields when you access them outside a transaction. Normally, access to a transient field causes `fetch()` or `dirty()` to be called to allow the `postInitializeContents()` and `preFlushContents()` methods to convert between persistent and transient states.

When you specify the `-noannotatefield` option, follow it with a qualified field name.

Avoiding finalize() Methods

strongly recommends that you do not define `java.lang.Object.finalize()` methods in application classes that are persistence capable. If your persistence-capable class must define a `finalize()` method, you must ensure that the `finalize()` method does not access any persistent objects. This is because the Java GC might call the `finalize()` method outside a transaction or from a thread that does not belong to the session of the object being finalized. Such a situation causes PSE Pro to signal `NoSessionException` and prevents execution of the `finalize()` method.

If your class defines a `finalize()` method, the class file postprocessor inserts annotations at the beginning of the `finalize()` method that change the persistent object to a transient object. This makes it safe to access fields of the finalized object. However, if the object has not been fetched, the fields are in an uninitialized state.

Troubleshooting Access to Persistent Objects

Incorrect program behavior can happen when your program does one of the following while an annotated method is executing:

- Aborts or commits a transaction
- Evicts one or all objects

The general result is that your program might incorrectly access additional persistent objects after the abort, commit, or eviction. The specific results vary according to the `retain` setting PSE Pro uses for the operation, as follows:

- `ObjectStore.RETAIN_STALE` should cause PSE Pro to signal `ObjectException` if your program tries to access a stale object. With the optimizations, your program might be able to access stale objects, which should not happen.
- `ObjectStore.RETAIN_HOLLOW`
  `ObjectStore.RETAIN_READONLY`
  `ObjectStore.RETAIN_TRANSIENT`
  `ObjectStore.RETAIN_UPDATE`

  These settings might cause your program to retrieve `null` or `0` values in place of correct values. Also, PSE Pro might fail to save some modifications in the database.

The class file postprocessor (`osjcfp`) uses three optimizations that can allow incorrect access to persistent objects. You can disable these optimizations by using the following `osjcfp` options:

- `-noarrayopt` disables optimization of `fetch()` and `dirty()` calls for array objects in looping constructs. This causes `osjcfp` to make the calls to `fetch()` or `dirty()` in every iteration rather than only in the first loop iteration.
- `-nothisopt` disables optimization of `fetch()` and `dirty()` calls for access to fields relative to `this` in nonstatic member methods. This causes `osjcfp` to insert a `fetch()` or `dirty()` call for each access to a field in `this`.
- `-noinitializeropt` disables optimization of `fetch()` and `dirty()` calls in constructors. Specify this option when you want the postprocessor to perform full annotation on constructors. When you specify this option, it applies to all classes that the postprocessor makes persistence capable.

Handling Unregistered Types

PSE Pro creates objects of type `UnregisteredType` when it must create a persistent object and it cannot find a class file for that object. The class might not be found because of a problem with the `CLASSPATH` or because the class is not available for a particular database.

If your application receives error messages that indicate unregistered types, the information here can help you determine what is happening and what to do about it. This section discusses

- How Can There Be Unregistered Types?
- Can Applications Work When There Are Types Not Registered?
- What Does PSE Pro Do About Unregistered Types?
- When Does PSE Pro Create UnregisteredType Objects?
- Can Your Application Run with UnregisteredType Objects?
- Troubleshooting ClassCastExceptions Caused by Unregistered Types
- Troubleshooting the Most Common Problem

How Can There Be Unregistered Types?

How can there be a type in the database with no corresponding `ClassInfo` subclass? This can happen when

- The `CLASSPATH` environment variable has been changed since the object was stored in the database and the class is no longer in the `CLASSPATH`.
- The `CLASSPATH` might include the directory or `.zip` or `.jar` file that contains the original class files, but not the directory or `.zip` or `.jar` file that contains the postprocessed class.
- The database includes an instance of a private class and there is not a corresponding `ClassInfo` subclass that describes that class. PSE Pro uses the reflection API to analyze persistence-capable public classes, but it is not available for private classes. Therefore, the `osjcfp` preprocessor creates a special subclass of `ClassInfo` for private classes that must be found whenever a private class is found in a database.

Can Applications Work When There Are Types Not Registered?

In some situations, it might not matter to your application that there is an object whose type is unregistered. For example, suppose you are looking up an element in a hash table. One of the elements in the hash table is of an unregistered type, but it is not the element you are looking for. Because PSE Pro creates an `UnregisteredType` object instead of signaling an exception, your application can keep running.

What Does PSE Pro Do About Unregistered Types?

PSE Pro provides the abstract class `UnregisteredType` to represent objects whose types are unregistered. When PSE Pro cannot find the `ClassInfo` subclass for a type that is referenced in your application, it

- Creates an `UnregisteredType` object to represent the type
- Uses the `UnregisteredType` object in place of the hollow object it would have created

You can never read or modify an `UnregisteredType` object. Because of this, it is important for you to understand

- When PSE Pro creates `UnregisteredType` objects
- Whether PSE Pro can use `UnregisteredType` objects in a particular situation

With this information, you can determine whether your application can run with objects of unregistered types. Your application can continue to run as long as you do not try to read or modify an `UnregisteredType` object.

When Does PSE Pro Create UnregisteredType Objects?

PSE Pro creates an `UnregisteredType` object when it encounters an object in a database and it determines that the type of that object is not registered.

PSE Pro encounters an object in a database when it

- Obtains the value of a database root
- Initializes an object and the value of one of the fields is a class, interface, or an array
- Initializes an array and the element type of the array is a class, an interface, or array
- Iterates over all objects in a segment

In the above list, *initialize* means to read the contents of the object out of the database and into the persistent Java object. This happens when PSE Pro calls `IPersistent.initializeContents()` and `IPersistent.postInitializeContents()`.

When PSE Pro encounters an object in a database, it determines whether there is already a Java object for the object in the database. If there is, PSE Pro uses that object. If there is not, PSE Pro checks to see whether the type of the object is registered.

If the type is not registered, PSE Pro tries to load the `ClassInfo` subclass for the type and register it. PSE Pro uses the regular Java class loading mechanism. Usually, this means that PSE Pro searches your `CLASSPATH`. Depending on the Java implementation you are using, Java class loading can also involve Java `ClassLoader` objects, as described in the *Java Language Specification*.

If PSE Pro cannot load the `ClassInfo` subclass, it cannot register the type and therefore it cannot create a hollow object for the type. In this case, PSE Pro creates a new Java object of type `UnregisteredType` and uses it in place of the hollow object.

Can Your Application Run with UnregisteredType Objects?

PSE Pro can use the `UnregisteredType` object if `java.lang.Object` is the type of the field in which the reference is being stored. For example, suppose you have the following class:

```
class Person {
  Pet mypet;
  Object mytrash;
}
```

You also have a database that contains one `Person` object. The value of the `Person.mypet` instance variable is an instance of the `Pet` class. The value of the `Person.mytrash` instance variable is an instance of the `Shoe` class.

Now suppose that the `Pet` class is an unregistered class. Your application opens the database and tries to read the `Person` object. This means that PSE Pro must initialize the `Person` object. When PSE Pro recognizes that the `Pet` class is unregistered, it creates an `UnregisteredType` object. PSE Pro then tries to assign the `mypet` instance variable to the `UnregisteredType` object. The code to do this is something such as the following:

```
mypet = (Pet)(handle.getClassField(1, XXX));
```

Typically, the postprocessor generates this code but you can specify it yourself in the `IPersistent.initializeContents()` method. In any case, the call to `handle.getClassField()` returns an `UnregisteredType` object. The cast to `Pet` is required because `Pet` is the type of the `mypet` instance variable. However, this cast does not work. You cannot cast an `UnregisteredType` to `Pet` because `UnregisteredType` is not `Pet` and is not a subclass of `Pet`. The Java VM signals a `ClassCastException` in the middle of the initialization. The `Person` object is never initialized.

Now suppose that the `Pet` class is registered and that the `Shoe` class, which is the type of the `Person.mytrash` instance variable, is not registered. PSE Pro creates an `UnregisteredType` object and the `handle.getClassField()` method returns it:

```
mytrash = (Object)(handle.getClassField(1, XXX));
```

This time, the cast works correctly because `UnregisteredType` is a subclass of `Object`. The initialization succeeds and the application continues to run.

Troubleshooting ClassCastExceptions Caused by Unregistered Types

If PSE Pro creates an `UnregisteredType` object and you do not try to do anything with it, your application should work well. Now suppose you try to do something with it. Because it exists, it must be in a variable of type `java.lang.Object`. (If it were not, you would have had trouble with it earlier, as in the `Pet` example in the previous section.)

You cannot do very much with objects of type `Object`, so it is likely that the first thing you would do is try to cast the `UnregisteredType` object to some specific type that you expect it to be. However, this does not work. If you try to cast an `UnregisteredType` object to a type other than `java.lang.Object` or `UnregisteredType`, the Java VM signals `ClassCastException`.

Unfortunately, the `ClassCastException` does not identify the type that is unregistered. There are two ways that you can determine the name of the type that is not registered:

- Change your program.
- Set the `com.odi.trapUnregisteredType` property.

Somewhere in your program, you have a variable of type `Object` whose value is an object of the `UnregisteredType` class. Modify your program to cast this variable to type `UnregisteredType`, then invoke the `getTypeName()` method on the `UnregisteredType` object. This returns the name of the type that is unregistered.

The disadvantage of this approach is that you must edit and recompile your code.

PSE Pro provides the `com.odi.trapUnregisteredType` property to help you determine the class that is unregistered. The default is that this property is not set, and it is usually best to use the default.

When PSE Pro determines that a type is not registered, it checks the setting of the `com.odi.trapUnregisteredType` property. If the property is not set (the default), PSE Pro creates an `UnregisteredType` object to represent the unregistered type. If `com.odi.trapUnregisteredType` is set, PSE Pro signals `FatalApplicationException` and provides a message indicating the name of the class that is unregistered.

The advantage of the `com.odi.trapUnregisteredType` property is that it provides the name of the class that is unregistered.

The disadvantage of the `com.odi.trapUnregisteredType` is that as soon as PSE Pro encounters the first object whose type is unregistered, your application stops running. If the object you want information about is the second object of an unregistered type that PSE Pro would encounter, PSE Pro never reaches that second object. When you set `com.odi.trapUnregisteredType`, PSE Pro signals `FatalApplicationException` as soon as it encounters the first object whose type is unregistered.

Troubleshooting the Most Common Problem

A common situation in which an `UnregisteredType` object signals `ClassCastException` occurs when you try to obtain a database root (`Database.getRoot()`) and the value of the root is an `UnregisteredType` object. For example:

```
Foo foo = (Foo) db.getRoot("foo");
```

If the `Foo` class is unregistered, the Java VM signals a `ClassCastException` when it comes to the `(Foo)` cast operation. See the previous section for two ways to determine the class that is unregistered in this situation.

However, when the value of a root is an unregistered type, it can mean that none of your persistence-capable types is registered. This is often true when an `UnregisteredType` object signals a `ClassCastException` very early in your program. Your best course of action is likely to be to ensure that your persistence-capable classes are in your `CLASSPATH` rather than trying to determine the class that is not registered.

Using Enums in Persistent Objects

In a PSE Pro database, an `Enum` can be a field in a persistent object or a field in a persistent array.

You cannot store an `Enum` as a top-level persistent object. In other words, an `Enum` cannot be a database root value, and it is not possible to explicitly migrate `Enum`s.

You cannot store an `Enum` directly in a `com.odi.util.OSTree*` collection object. If you try to, PSE Pro signals an `IllegalArgumentException`. However, you can store an object that contains an `Enum` in an `OSTree*` object.

An `Enum` constant must use the ASCII character set. If you need to store `Enum` constants that use characters outside the ASCII character set, contact Progress technical support.

While an `Enum` can implement an interface, your PSE Pro application must not try to store an `Enum` in a field that you declare to be an interface. Trying to do so might corrupt your database.

In your application, you can change the definition of an `Enum` without upgrading or evolving a database that contains instances of that `Enum`. This is because PSE Pro does not validate the schema of an `Enum`. The schema of an `Enum` is the list of constants and their declared order. Consequently, an application can change the definition of an `Enum` without incurring a PSE Pro schema validation exception. If an application tries to read an `Enum` constant that you removed from the `Enum`'s definition, PSE Pro signals an `EnumConstantNotPresentException`.

You can create a query optimization index based on an `Enum` field. PSE Pro orders such an index in the order in which you declared the possible values for the `Enum`, which is the value returned by the `java.lang.Enum.ordinal()` method. If you

modify the schema of an `Enum` after you use it to create an index, it is your responsibility to update the index.

You do not need to postprocess `Enum`s for persistence. However, if a method in an `Enum` accesses a persistent field of a persistent class, you must postprocess the `Enum` to make it persistence-aware.

Using Generic Types

You can define a class that contains one or more fields of a generic type and then store an instance of that class in a PSE Pro database. You can also store a generic class in a PSE Pro database.

Because of the way that Java implements generic types, the schema information that PSE Pro maintains does not include generic type information. For example, PSE Pro considers the following class definitions to be identical:

```
public class Department {
   OSVector employees;
}
```

```
public class Department {
   OSVector<Employee> employees;
}
```

Consequently, PSE Pro schema validation does not validate information supplied by the generic types. This means, for example, that PSE Pro cannot prevent you from accidentally storing a non-`Employee` object in the `employees` collection.

Chapter 7
Working with Collections

PSE Pro provides a set of persistence-capable utility collections classes in the `com.odi.util` package. These classes rely on the interfaces defined in the `java.util` package.

This chapter discusses the following topics:

Description of PSE Pro Utility Collections

PSE Pro provides a number of utility collections interfaces and classes in the `com.odi.util` package. In addition, PSE Pro provides a query facility in the `com.odi.util.query` package. PSE does not include the query facility.

A collection is an object that groups together other objects. It provides an effective means of storing and manipulating groups of objects and supports operations for inserting, removing, and retrieving elements.

Collections form the basis of the PSE Pro query facility, which allows you to select those elements of a collection that satisfy a specified condition. However, some collections can be queried and others cannot. Therefore, before you create a collection and store it in a database, you should consider how you plan to use a collection. When you know what you need, you can select the best persistent collection representation for your application.

To introduce you to the PSE Pro utility collections facility, this section discusses the following topics:

- Introduction to java.util Interfaces and Classes
- Description of OSHashBag
- Description of OSHashMap

- Description of OSHashSet
- Description of OSHashtable
- Description of OSTreeMapxxx
- Description of OSTreeSet
- Description of OSVector
- Description of OSVectorList
- Advantages of Using PSE Pro Utility Collections
- Background About Utility Collections and Java Collections

Introduction to java.util Interfaces and Classes

The `java.util.Collection` and `java.util.Map` interfaces provide methods for operating on PSE Pro collections.

- `Collection` provides methods for operating on groups of objects in which the objects might be ordered, might be duplicated, and can be queried. The internal representation of a class that implements `Collection` might be a hash table, a binary tree, or another data structure.
  - The `java.util.List` interface extends `java.util.Collection`. In collections that implement `List`, the elements are ordered and duplicates are allowed.
  - The `java.util.Set` interface extends `java.util.Collection`. In collections that implement `Set`, the elements are not ordered and duplicates are not allowed.
- `Map` provides methods for operating on groups of key/value entries. Each key can map to at most one value. You cannot query collections that implement `Map`.

The PSE Pro utility collections facility provides the persistence-capable `java.util` classes shown in the following table. Most of these classes implement a `java.util` interface (many implement other interfaces as well).

Class	*Implements*
`OSHashBag`	`Collection`
`OSHashMap`	`Map`
`OSHashSet`	`Set`
`OSHashtable`	None
`OSTreeMapByteArray`	`Map`
`OSTreeMapDouble`	`Map`
`OSTreeMapFloat`	`Map`
`OSTreeMapInteger`	`Map`
`OSTreeMapLong`	`Map`
`OSTreeMapString`	`Map`
`OSTreeSet`	`Set`
`OSVector`	`Collection`
`OSVectorList`	`List`

Not necessary to postprocess classes

You need not postprocess the classes in the utility collections facility. They are already persistence capable. If you define a subclass that extends any of these classes and you want the subclass to be persistence capable, you must either run the postprocessor on the subclass or manually annotate the subclass.

Example

The `query` demo provides an example of using PSE Pro with utility collections. See the `README` file in the `com/odi/demo/query` directory.

Hash code

The Java collections interfaces specify the behavior of the `hashCode()` method on instances of the `Set`, `Map`, and `List` types. This `hashCode()` specification is based on the contents of the collection; the `hashCode` of a collection changes depending on the elements that are added or removed. This means that it is not advisable to store an instance of a set, map, or list class in a hash table unless the set or list is immutable and will never change.

Description of OSHashBag

An `OSHashBag` is an unordered collection that allows duplicates. `OSHashBag`s not only keep track of what their elements are but also of the number of occurrences of each element. As the name implies, a hash table is the internal representation for an `OSHashBag`. `OSHashBag` directly implements the `java.util.Collection` interface, so you can query instances of `OSHashBag`.

Description of OSHashMap

An `OSHashMap` is a map that allows duplicate values but not duplicate keys. Unlike `OSHashBag`, `OSHashMap` associates a key with each value in the map. When you insert a value into an `OSHashMap`, you specify the key along with the value. You can retrieve a value with a given key. The internal representation of an `OSHashMap` is a hash table. `OSHashMap`s do not allow null keys or null values.

Because `OSHashMap` implements the `Map` interface rather than the `Collection` interface, you cannot query `OSHashMap`s. However, you can query the collection views of a map: `Map.keySet()`, `Map.values()`, and `Map.entrySet()`. See Querying Collection Views of Map Entries on page 146.

The `OSHashMap.equals()` method performs value (contents) comparisons, as described by `Map.equals()`, to determine whether two `Map`s are equal. This is the only difference between `OSHashMap` and `OSHashtable`. The `OSHashtable.equals()` method compares the identities of the two objects to determine equality. The `OSHashtable.hashcode()` method generates a hash code based on object identity; it is not based on the contents of the `OSHashtable`. For information about content comparisons and identity comparisons, see “OSHashtable and OSVector” on page 147.

Description of OSHashSet

An `OSHashSet` is an unordered collection that does not allow duplicates. If you try to insert a value into an `OSHashSet` and the set already contains that value, the set remains unchanged. `OSHashSet` implements the `java.util.Set` interface. As its name implies, a hash table is the internal representation of an `OSHashSet`. Because `OSHashSet` indirectly implements `java.util.Collection`, you can query `OSHashSet`s.

`OSTreeSet`s are capable of storing much larger persistent collections than `OSHashSet`s. However, `OSTreeSet`s must be persistent; it is not possible to create a transient instance of an `OSTreeSet`. If your collection is small, an `OSHashSet` is the better choice. If your collection is large, an `OSTreeSet` performs better.`OSTreeSet` is only available in PSE Pro.

Description of OSHashtable

An `OSHashtable` is also an unordered collection that allows duplicates. This class has the same APIs as `java.util.Hashtable`.

`OSHashtable` associates a key with each element. When you insert an element into an `OSHashtable`, you specify the key along with the element. You can retrieve an element with a given key. While the internal representation of an `OSHashtable` is a hash table, it is a map-like structure.

Because `OSHashtable` does not implement the `java.util.Collection` interface, you cannot query `OSHashtable`s. However, you can query the collection views of an `OSHashtable`. See Querying Collection Views of Map Entries on page 146.

The `OSHashtable.equals()` and `OSHashtable.hashCode()` methods perform reference (identity) comparisons and not value (contents) comparisons. This is the only difference between `OSHashtable` and `OSHashMap`. The `OSHashMap` methods perform content comparisons. For information about content comparisons and identity comparisons, see “OSHashtable and OSVector” on page 147.

By default, an `OSHashtable` allocates room for 50 elements. You can presize an `OSHashtable` to better match what your application needs. In addition, you can delay allocation of `OSHashtable` substructure, which PSE Pro uses to represent the `OSHashtable` until elements are actually added to the `OSHashtable`. To do this, specify the `lazy` argument to the `OSHashtable` constructor, as follows:

```
OSHashtable(int initialBufferSize, int capacityIncrement,
  boolean lazy)
```

Description of OSTreeMap*xxx*

`OSTreeMap` is only available in PSE Pro. It is based on a B-tree representation that is tuned for large persistent collections. `OSTreeMap` is an abstract class with several

concrete subclasses. In all OSTreeMap*xxx* instances, the values are objects. Each subclass uses a different type for keys, as shown in the following table:

Class	*Key Type*
OSTreeMapByteArray	ByteArray
OSTreeMapDouble	Double
OSTreeMapFloat	Float
OSTreeMapInteger	Integer
OSTreeMapLong	Long
OSTreeMapString	String

An OSTreeMap*xxx* is a map that allows duplicate values but not duplicate keys. Each OSTreeMap*xxx* associates a key with a value in the map. When you insert a value into an OSTreeMap*xxx*, you specify the key along with the value. You can retrieve a value with a given key. OSTreeMap*xxx*s do not allow null keys or null values.

The OSTreeMap*xxx* classes extend OSTreeMap, which implements java.util.Map. Consequently, you cannot query OSTreeMap*xxx*s. However, you can query the collection views of a map: Map.keySet(), Map.values(), and Map.entrySet(). See Querying Collection Views of Map Entries on page 146.

The OSTreeMap*xxx* classes are designed for large persistent aggregations. These classes allow you to iterate over the collection or query the collection without fetching any objects from the database except those that are explicitly returned to you. PSE Pro does not create hollow objects to represent the elements until they are fetched, thus reducing Java heap overhead when a subset of OSTreeMap is accessed. OSTreeMap collections can only be persistent.

Description of OSTreeSet

An OSTreeSet is an unordered collection that does not allow duplicates. If you try to insert a value into an OSTreeSet and the set already contains that value, the set remains unchanged. OSTreeSet implements the java.util.Set interface. As its name implies, a balanced tree is the internal representation of an OSTreeSet. Because OSTreeSet indirectly implements com.odi.util.IndexedCollection, which extends java.util.Collection, you can query OSTreeSets.

The OSTreeSet class is designed for very large persistent aggregations. This class allows you to iterate over the collection or query the collection without fetching any objects from the database except those that are explicitly returned to you. PSE Pro does not even create hollow objects to represent the elements. OSTreeSet collections can only be persistent.

recommends that if you are going to query a collection that contains a particularly large number of objects, you should define the collection as an OSTreeSet or a subclass of OSTreeSet. OSTreeSet is the only collections class for which PSE Pro provides the ability to add indexes. Indexes can speed up queries on very large

collections. For more information, see Enhancing Query Performance with Indexes on page 161.

Primary index

`OSTreeSet` has a constructor that allows you to create an empty `OSTreeSet` that has a primary index. The method signature is

```
public OSTreeSet(Placement place,
  Class primaryIndexElementType,
  String primaryIndexPath)
```

A primary index is used for queries and for looking up objects in the `OSTreeSet`. The primary index must contain no duplicate keys and must contain all elements in the `OSTreeSet`. The benefits of a primary index include

- Faster look-up times for objects in some cases
- Faster insertion and removal of objects from the set
- An iterator that returns objects in their primary key order
- Less storage space used when compared to an `OSTreeSet` with a nonprimary index.

  A primary index saves storage space *only* if an existing index is designated as a primary index. If you need to create an additional index in order to designate a primary index, then no storage space is saved unless you wanted the index anyway.

  Storage space is saved when an index is designated as a primary index because the hashed-based map used to locate objects is removed from the `OSTreeSet`.

Primary index maintenance

If you decide to use a primary index, you must ensure that your application performs index maintenance when modifying fields of objects that affect the primary index. Otherwise, the contents of the `OSTreeSet` are not maintained and methods such as `OSTreeSet.add()`, `OSTreeSet.contains()`, and `OSTreeSet.remove()` will not work correctly. For more information on index maintenance, see Managing Indexes and Index Values on page 166.

Adding a primary index

To designate that an existing `OSTreeSet` index be used as a primary index, call the following method:

```
public void setPrimaryIndex(Class elementType, String path)
```

PSE Pro signals `IndexException` if the specified index is not found or allows duplicate key values, or if the `OSTreeSet` contains elements that are not instances of the *`elementType`*.

Obtaining a primary index

To obtain a primary index from an `OSTreeSet`, call the following method:

```
public IndexMap getPrimaryIndex()
```

The primary index is returned if one exists; otherwise, `null` is returned.

Specifying no primary index

To specify that none of the existing indexes of an `OSTreeSet` be used as a primary index, call the following method:

```
public void noPrimaryIndex()
```

When you specify no primary index, you are not removing the index itself. The index is still available for queries. It just means that PSE Pro will use a map of object hash codes to find objects in the set instead of using the primary index. This method does nothing if there is no primary index on the set.

Comparing OSTreeSet and OSHashSet

The primary difference between `OSTreeSet` and `OSHashSet` is the internal representation. Also `OSTreeSet` supports indexes, whereas `OSHashSet` does not. `OSTreeSet`s can only be persistently allocated. It is not possible to create a transient `OSTreeSet`. `OSTreeSet` is available in PSE Pro, but not in PSE. For more information on which collection to use, see The Way to Choose a Collection on page 148.

Exported objects

The `OSTreeSet` class has a constructor for creating exported objects.

Description of OSVector

An `OSVector` is a persistent expandable array that implements `java.util.Collection`. You can query `OSVector`s.

An `OSVector` associates each element with a numerical position based on insertion order. By default, `OSVector`s allow duplicates. In addition to simple insert (insert into the beginning or end of the collection) and simple remove (remove the first occurrence of a specified element), you can insert, remove, and retrieve elements based on a specified numerical position or based on a specified iterator position. An `OSVector` does not have quick look-up by object or key. Therefore, the overhead for an `OSVector` is lower than for utility collections that have quick look-up.

The `OSVector.equals()` and `OSVector.hashCode()` methods perform reference (identity) comparisons and not value (contents) comparisons. This is one difference between `OSVector` and `OSVectorList`. The `OSVectorList` methods perform content comparisons. For information about content comparisons and identity comparisons, see “OSHashtable and OSVector” on page 147.

By default, an `OSVector` allocates room for 32 elements. You can presize an `OSVector` to better match what your application needs. In addition, you can delay allocation of `OSVector` substructure, which PSE Pro uses to represent the `OSVector` until elements are actually added to the `OSVector`. To do this, specify the `lazy` argument to the `OSVector` constructor, as follows:

```
OSVector(int initialBufferSize, int capacityIncrement, boolean lazy)
```

Description of OSVectorList

An `OSVectorList` is a collection that implements a persistent expandable array. It implements the `java.util.List` interface and functions exactly like an `OSVector`, except in the following way.

The `OSVectorList.equals()` and `OSVectorList.hashCode()` methods perform value (contents) comparisons and not reference (identity) comparisons. This makes `OSVectorList` unsuitable for storage in a persistent hash table or any other hash-table-based collection representation. The `OSVector` methods perform identity comparisons. For information about content comparisons and identity comparisons, see “OSHashtable and OSVector” on page 147.

Advantages of Using PSE Pro Utility Collections

The advantages of using `com.odi.util` interfaces and classes are as follows:

- The interfaces and classes in `com.odi.util` rely on and extend the interfaces defined in Java.
- The classes are persistence capable.
- There are collection representations that support queries.
- Some of the classes — `OSTreeMap`*xxx* and `OSTreeSet` – support very large aggregations.

Querying Collection Views of Map Entries

The `OSHashMap` and `OSTreeMap`*xxx* classes extend `java.util.Map` and not `java.util.Collection` and, therefore, you cannot use the PSE Pro query facility on them. However, each of the classes that implements `Map` defines the following methods:

- `keySet()` returns a `java.util.Set` view of the keys contained in the map.
- `values()` returns a `java.util.Collection` view of the values contained in the map.
- `entries()` returns a `java.util.Set` view of the key/value mappings contained in the map.

The `OSHashtable` class, although it does not implement `Map`, also defines these methods.

You can use the PSE Pro query facility to query the `Collection` and `Set` views returned by the `keySet()`, `values()`, and `entries()` methods.

Transient views

While `OSHashtable`, `OSHashMap`, and the `OSTreeMap`*xxx* subclasses are persistence capable, the views returned by the `entries()`, `keySet()`, and `values()` methods are not. These are transient views of persistence-capable classes.

Background About Utility Collections and Java Collections

Following is background information about how the PSE Pro utility collections fit with the Java collections. This discussion assumes that you are familiar with the Java collections API.

PSE Pro provides a collections package that relies on and extends the `java.util` collections. In addition, PSE Pro includes querying and indexing facilities. The new collections implementations are in the `com.odi.util` package.

The core collections interfaces defined in the `java.util` package are

- `java.util.Collection`
- `java.util.Set`
- `java.util.List`
- `java.util.Map`

In the Java collections, classes and behaviors are based on these interfaces. Consequently, you can usually use any representation that is parallel to a particular interface. The `java.util` implementations and their corresponding PSE Pro implementations are shown in the following table:

Interface	*java.util Class*	*PSE Pro Class*
`Collection`	None	`com.odi.util.OSHashBag`
`Set`	`java.util.HashSet`	`com.odi.util.OSHashSet`
`Set`	`java.util.ArraySet`	`com.odi.util.OSTreeSet`
`List`	`java.util.Vector`	`com.odi.util.OSVector`
`List`	`java.util.ArrayList`	`com.odi.util.OSVectorList`
`List`	`java.util.LinkedList`	None
`Map`	`java.util.Hashtable`	`com.odi.util.OSHashtable`
`Map`	`java.util.HashMap`	`com.odi.util.OSHashMap`
`Map`	`java.util.ArrayMap`	None
`Map`	`java.util.TreeMap`	`com.odi.util.OSTreeMap`*xxx*

OSHashtable and OSVector

`com.odi.util.OSHashtable` and `com.odi.util.OSVector` have been updated to be parallel to most of the Java specifications. They do not quite meet the description of the Java behavior for `equals()` and `hashCode()`. The JDK 1.2 changed this behavior in an incompatible way for these two classes.

The Java `List`, `Set`, and `Map` interfaces mandate an `equals()` method that does value comparison and not reference comparison. That is, two `Set`s are equal if they have the same elements, two `List`s are equal if they have the same elements in the same order, and two `Map`s are equal if they have the same key/value pairs.

This places corresponding constraints on the `hashCode()` method because `(a.equals(b)) => (a.hashCode()==b.hashCode())`. The PSE Pro `OSHashtable` and `OSVector` classes, however, implement persistent (unchanging) `hashCode`s and rely on `Object.equals()`. The JDK definition for `hashCode` means that classes that meet the Java specification should not be stored in hash tables because their `hashCode`s change when elements are added or removed. For these two classes, PSE Pro retains the old identity-based definitions rather than moving to the new content-based definitions of `equals()` and `hashCode()`.

Collection interface

There are no concrete implementations of the `Collection` interface in Java. `Collection` is essentially a `Bag`, that is, a `Set` that might contain duplicates. PSE Pro includes the `com.odi.util.OSHashBag` and `com.odi.util.OSVector` classes to implement `Collection`.

The Way to Choose a Collection

Your choice of how to implement a collection depends on

- The amount of data to be stored in the collection

  The following numbers can help you determine the type of collection to use. The efficiency of a collection is based on its performance and storage size.

  - Java arrays are efficient for up to 1000 elements.
  - `OSHashtable`s are efficient for small collections that have fewer than 1000 elements.
  - `OSVectors` are efficient for medium collections that have as many as 1,000,000 elements.
  - `OSTreeMap`s and `OSTreeSet`s are efficient for medium and large collections that have between 200 and several million elements.

  The recommended size for `OSVector` overlaps with the sizes for other collection types. When sizes overlap, you should use `OSVector` when the keys are contiguous integers.

  The recommended size for `OSHashtable` overlaps with the size for `OSTreeMap`. When the sizes overlap, use an `OSHashtable` only if the same element is accessed multiple times within a transaction.

- Whether the queries you use can benefit from using an index

  Indexes are useful when you are querying a collection that is larger than a few hundred elements and when you know in advance what fields will be queried.

  `OSTreeSet` is the only collection that can have indexes, one of which you can designate as a primary index. If an `OSTreeSet` has a primary index, that index is used for determining set membership. With a primary index, inserting and removing elements is faster when compared to a set with a nonprimary index, less storage space is used, and the iterator can return elements in their primary key order.

- Your familiarity with a third-party library

  You might want to use a particular library because you already know how to use it.

- Features required by your application

- Importance of compatibility with the Java collection interfaces

  The `OSHashtable` class is not compatible with the Java `Map` interface because its `.equals` operator is identity based and its hash code is not based on contents. All other collections in `com.odi.util` are compatible with the Java API.

- Importance of compatibility between OSJI and PSE Pro

  PSE Pro does not have the `com.odi.coll` collections.

Comparing Collection Classes

To help you choose the right persistent collection representation for your application, the following table compares the behavior of the different collection classes in `com.odi.util`.

Table note

In the following table

- The key in an `OSVector` and `OSVectorList` is the offset.
- Elements in an `OSTreeSet` are in key value order *only* when there is a primary index.

Class	*Elements in Key Value Order?*	*Duplicate Keys Allowed?*	*Duplicate Values Allowed?*	*.contains (Object) Slow?*	*.equals() Compares By*	*Queries Allowed?*	*Size*
`OSHashBag`	No	Yes	Yes	Yes	Identity	Yes	Small
`OSHashMap`	No	No	Yes	Yes	Content	No	Small
`OSHashSet`	No	N/A	No	No	Content	Yes	Small
`OSHashtable`	No	No	Yes	Yes	Identity	No	Small
`OSTreeMap`*xxx*	Yes	No	Yes	Yes	Content	No	Medium/Large
`OSTreeSet`	No/Yes	N/A	No	No	Content	Yes	Medium/Large
`OSVector`	Yes	No	Yes	Yes	Identity	Yes	Medium
`OSVectorList`	Yes	No	Yes	Yes	Content	Yes	Medium

Performance-Based Recommendations for Collections

This section lists additional performance recommendations that you might want to consider when choosing which collection type to use.

OSHashtable

The `OSHashtable` becomes *very* inefficient when the collection grows larger than a few thousand elements. The `OSHashtable` is faster than an `OSTreeSet` *only* when the same set of elements is accessed many times within the same transaction.

OSTreeMap*xxx* and OSTreeSet

The `OSTreeMap`*xxx* and `OSTreeSet` implementations use an approach that requires the fewest object materializations for accessing their elements. As a result, `OSTreeMap`*xxx* and `OSTreeSet` implementations have the fastest initial access to elements, but they are not as fast as the other types of collection classes when you are accessing the elements for the second time in a transaction.

OSTreeSet

Designating one of the indexes of an `OSTreeSet` to be a primary index makes inserting and removing elements faster.

com.odi.coll package

The collections in the `com.odi.coll` package are approximately the same as the `com.odi.util` collections in their look-up speed for an object, but the `com.odi.coll` collections are much slower when you are inserting and iterating through elements.You should use the `com.odi.coll` collections *only* if your Java applications will access C++ code.

Using PSE Pro Utility Collections

To help you use PSE Pro utility collections, this section discusses the following topics:

- Creating Collections
- Navigating Collections with Iterators
- Performing Collection Updates During Iteration

Creating Collections

Each collection representation has one or more constructors that you can use to create collections. For example:

```
Database db = Database.create(args[1], ALL_READ | ALL_WRITE);
Transaction.begin(UPDATE);
db.createRoot("collection", new OSTreeSet(db));
Transaction.current().commit();
```

For details about each class's constructors, see the *ObjectStore Java Interface API Reference*.

Navigating Collections with Iterators

The `Iterator` and `ListIterator` interfaces help you navigate within a utility collection. An *iterator* is an instance of the `java.util.Iterator` or `java.util.ListIterator` interface. It designates a position in a collection. You can use iterators to traverse collections as well as to remove elements from collections.

With the JDK 1.2, `Iterator` took the place of `Enumeration`. `Iterator` provides the same capabilities as `Enumeration` (though method names are different), and it allows you to remove elements from the underlying collection.

The `ListIterator` interface extends the `Iterator` interface. A class that supports traversal by `ListIterator` must also implement `List`. The additional methods that `ListIterator` provides allow you to

- Insert objects relative to the current position of the iterator
- Traverse the list in reverse as well as forward
- Replace an element in the underlying list
- Retrieve the index of an element

The `IndexIterator` interface in `com.odi.util` extends `java.util.Iterator` and allows you to traverse an index or map structure. You can use the `IndexIterator` interface to obtain the key and value for elements in the underlying collection.

Performing Collection Updates During Iteration

While you are iterating through a collection, you can use the `Iterator` and `ListIterator` interface methods to modify that collection. This assumes that the implementation of the `Iterator` or `ListIterator` interface supports the methods that modify underlying collections. (Java defines some of these methods as optional.

You should check the API reference information for the particular class you are using to determine exactly the behaviors that are supported.)

When a thread is iterating over a collection, that thread and cooperating threads can modify the object returned by the iteration. If you are using an `Iterator`, your application cannot add elements to the collection or change the order of the collection. If you are using a `ListIterator`, your application can only use `ListIterator` methods to modify the collection.

Suppose you do add an element in the middle of an iteration, and then try to use the same iterator. PSE Pro recognizes that the collection has been modified and signals `ConcurrentModificationException`. At this point, if you create a new iterator, it recognizes the updated collection and does not signal an exception.

Querying PSE Pro Utility Collections

The `com.odi.util.query.Query` class provides a mechanism for querying collections objects that implement the `java.util.Collection` interface. A query applies a predicate expression (an expression that evaluates to a `boolean` result) to all elements in a collection. The query returns a subset collection of all elements for which the expression is true. You can query the following classes that implement the `Collection` interface:

- `OSHashBag`
- `OSHashSet`
- `OSTreeSet`
- `OSVector`
- `OSVectorList`

To accelerate the processing of queries on particularly large collections, you can build indexes on the collection. For information about indexes, see the next section, Enhancing Query Performance with Indexes on page 161.

This section provides the following information about queries on PSE Pro utility collections:

- Creating Queries
- Description of Query Syntax
- Sample Program That Uses Queries
- Matching Patterns in Query Strings
- Using Free Variables in Queries
- Executing Queries
- Limitations on Queries

prolite.jar

If you are using PSE Pro, you must have `pro.jar` and not `prolite.jar` in your `CLASSPATH` if you want to use queries and indexes. The `prolite.jar` file does not include the PSE Pro query facility.

Creating Queries

To create a query, run the `com.odi.util.query.Query` constructor and pass in a `Class` object and a query string. Following is the constructor:

```
public Query(Class elementType, String queryExpression)
```

There is also a constructor that allows you to specify a `FreeVariables` map.

The *`elementType`* class or interface provides the context in which the query facility interprets *`queryExpression`*. This must be a publicly accessible class or interface. When your application calls the `Query.select()` or `Query.pick()` method to execute the query against a particular collection, every element of that collection must be an instance of (in the sense of `instanceof`) the *`elementType`* that was specified when the query was created. Any element of the collection that is not an instance of *`elementType`* is not returned in the query result, even if it evaluates to `true` for the predicate.

The *`queryExpression`* is a predicate (that is, an expression with a `boolean` result) that the query facility evaluates on each element of the collection. The *`queryExpression`* operands can be literals and names.

Literals can be of any of the Java primitive types, including the special values `true`, `false`, and `null`. Because the query expression is a `String`, you must enclose any embedded strings in escaped quotation marks, like `\"this\"`.

Names can consist of a single identifier or they can consist of a sequence of identifiers separated by periods. Names can be either free variables or member names (field or method names). You must explicitly specify free variables in the *`freeVariables`* argument of the three-argument `Query` constructor. Any name that is not a free variable is interpreted as a member name.

Member accesses are interpreted as accessing public members, including static members of an object of class/interface *`elementType`*, if possible. This interpretation works as though there were an implicit `this` argument of *`elementType`* at the root of the name expression. Any member access that cannot be interpreted as a member access on *`elementType`* is interpreted as a static access. Static accesses are resolved as if the package containing *`elementType`* were imported.

Queries can contain methods that take arguments. The arguments can be literals or bound variables.

Example

For example, to define a simple query:

```
Query q = new Query(Employee.class, "salary < 50000");
```

The query expression can refer to classes without specifying a package name. PSE Pro treats the query expression as if it were defined in a file in another package that has imported the package of the `Class` object that was passed to the `Query` constructor. This default package matters only for class names, though, not for member access. Only public classes and members are accessible within the query.

An application can run the example query on a specific collection with a call to the `Query.select()` method that specifies the collection to be queried as the argument. For example:

```
Query q = new Query(Employee.class, "salary < 50000");
Collection employees = (Collection)db.getRoot("employees");
Set result = q.select(employees);
```

When you create a query, you do not bind it to a particular collection. You can create a query, run it once, and throw it away. Alternatively, you can reuse a query multiple times against the same collection, perhaps with different bindings for free variables, or against different collections.

If the syntax of your query is wrong, `QueryException` is thrown at the point at which you create the query. You need not wait for the application to optimize or to execute the query. However, the query facility cannot detect incorrect free variable bindings until you specify them when you execute the query on a collection.

Sample program

The following sample program uses a query that takes a method as an argument. To run this program, you need to

1. Compile it.

   ```
   javac QueryMethodWithArgs.java
   ```

2. Use the postprocessor to make the class persistence capable.

   ```
   osjcfp -dest . -inplace QueryMethodWithArgs.class
   ```

3. Run it as an application with no arguments.

   ```
   java QueryMethodWithArgs
   ```

```
import com.odi.*;
import com.odi.util.*;
import com.odi.util.query.*;

/**
  * An example of performing queries that use methods as
  * arguments. Instances of this class have first name and last
  * name fields. The getName(useFirst) method returns the first
  * or last name, depending on the value of the useFirst
  * argument.
  * You can use an index on the getName() method with an
  * appropriate value when evaluating the query, because its
  * argument is a constant value. */

public class QueryMethodWithArgs implements ObjectStoreConstants {

  /* Fields */

  String first;
  String last;

  /**
  * Constructor.
  */

  QueryMethodWithArgs(String first, String last) {
    this.first = first;
    this.last = last;
  }

  /**
  * Returns the first name if useFirst is true, otherwise the
```

```
    last name.
  */

  public String getName(boolean useFirst) {
    return useFirst ? first :last;
  }

  /**
  * Include the field values in the print string.
  */
  public String toString() {
  return "QueryMethodWithArgs{ first = " + first + ", last =
    " + last + "      "}" ;

  /**
  * Main routine to run application.
  */
  public static void main(String[] args) {
    Session session = Session.create(null, null);
    try {
    session.join();

    Database db = Database.create("foo.odb", 0664);

  Transaction.begin(UPDATE);

  /* Create an OSTreeSet. */
  OSTreeSet set = new OSTreeSet(db);
  db.createRoot("set", set);

  /* Add two objects to it. */
  set.add(new QueryMethodWithArgs("John", "Doe"));
  set.add(new QueryMethodWithArgs("Jane", "Doe"));

  /* Add an index on getName(false), which is an index on the
    last names of the objects. */
  set.addIndex(QueryMethodWithArgs.class, "getName(false)");

  Transaction.current().commit();

  Transaction.begin(READONLY);

  set = (OSTreeSet)db.getRoot("set");

  /* Create a query to look for last names equal to "Doe". */
  Query query = new Query(QueryMethodWithArgs.class,
      "getName(false) == \"Doe\"");

  /* Perform the query. */
  Iterator iterator = query.iterator(set);

  /* Print the matches. */
  while (iterator.hasNext())
  System.out.println(iterator.next());

  } finally {
  session.terminate();
  }
  }
}
```

Description of Query Syntax

PSE Pro performs syntax analysis of the query expression in the context of the `elementType` class or interface that is passed to the query constructor. This must be a publicly accessible class or interface, or a derived type.

When the query is executed against a particular collection using the `select()` or `pick()` method, every element of that collection must be an instance (in the sense of `instanceof`) of the `elementType` that was specified when the query was created.

The `queryExpression` is a predicate. The query is executed on a collection by evaluating this query expression on each element of the collection. However, it might not be necessary to explicitly fetch and examine all elements of the collection. This depends on the available indexes and query optimization strategy.

Supported operations

Queries on utility collections can include most Java operations, as follows:

- Arithmetic: `+ / - * %`
- Bitwise: `^ | &`
- Unary numeric: `~ -`
- Unary logic: `!`
- Relational: `> < <= >= instanceof`
- Equality: `== !=`
- String concatenation: `+`
- Conditional AND, OR: `&& ||`
- Shift operations: `<< >> >>>`
- Cast operations: (*type*)

Unsupported operations

The following operations are not supported:

- Assignment: `= += *= /= %= -= <<= >>= >>>= &= ^= |=`
- Conditional: `?:`
- Array dereference: `[]`
- New: `new`
- Prefix/Postfix: `++ --`

Statements are not permitted. Only expressions are permitted.

For details on operations and the operands, see the *Java Language Specification*. The operators have their usual Java meanings except for the relational and equality operators when used with `String` operands. In a query expression, PSE Pro uses these operators to compare the contents of the two strings rather than their identities. Null `String`s are considered to be less than all other values.

String literals

In a query expression, you must enclose `String` literals in escaped quotation marks. For example:

```
new Query(Foo.class, "name == \"Davis\"")
```

You can specify wildcards in query strings. You can search for substrings and perform case-insensitive searches. See Matching Patterns in Query Strings on page 156.

Wrapper objects

The query facility treats wrapper objects just as it does other `Objects`. For example, suppose you have the query expression `"A==B"`. `A` and `B` refer to `Integer` wrappers. This results in an identity check on the objects. The query facility determines whether `A` and `B` both refer to the same wrapper instance. The query facility does not check that the values of `A` and `B` are equal. You can specify `"A.intValue()==V.intValue()"` to compare contents.

This behavior might change in a future release so that the query facility treats wrapper objects in the way that it treats primitives. Consequently, you should not rely on the identity check for wrapper objects.

Other rules

You can use parentheses to group expressions.

The precedence and associativity of the operators is the same as that for the Java language.

The entire query expression must resolve to a Boolean value.

Sample Program That Uses Queries

In the `com/odi/demo/query` directory, there is a sample program that uses PSE Pro utility queries. See the `README.htm` file in that directory.

Matching Patterns in Query Strings

Specifying a pattern-matching query

To specify a string pattern to be matched in a query, the pattern matching operator (~~) is used. This operator, which has greater precedence than the multiplication operator (*), has two arguments. These arguments must be either `String`s or null. The left-hand argument specifies the text to be checked for a match. The right-hand argument specifies the pattern to be matched.

Pattern-matching characters

The following characters have special meanings when used in the right-hand argument of the pattern matching operator. All other characters match themselves.

Operator	*Function*
?	Matches any single character
*	Matches 0 or more of any character
&	Escape character
[	Reserved
]	Reserved
(	Reserved
)	Reserved
|	Reserved

Note

The reserved characters are invalid if they are not preceded by an ampersand (&).

The following table shows special two-character sequences, known as escape sequences, that start with an ampersand (&). These escape sequences are used to include characters literally in the pattern without their special meaning and to enable case-insensitive matching.

Note that the ampersand (&) must appear in front of every sequence. An ampersand followed by any other character is invalid.Case sensitivity in matching

Escape Sequence	*Function*
&?	Matches a question mark
&*	Matches an asterisk
&[	Matches left square bracket
&]	Matches right square bracket
&(	Matches left parenthesis
&)	Matches right parenthesis
&|	Matches a vertical bar
&&	Matches an ampersand
&i	Enables case-insensitive matching

By default, pattern matches are case sensitive. The &i escape sequence enables case-insensitive matching for an entire pattern. This escape sequence can be specified only at the start of a pattern.

Optimizing pattern matching

The pattern-matching operator takes advantage of any ordered indexes available on the text being matched. If the pattern starts with a character other than an asterisk (*) or a question mark (?), the query searches only the portion of the index that matches the initial constant prefix. Therefore, patterns that specify a constant prefix produce much more efficient queries.

Pattern-matching examples

The following pattern-matching examples use the following class:

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

- Matching a name beginning with the characters `Tom`:

```
new Query(Person.class,"name ~~ \"Tom*\"");
```

- Matching a name ending with the characters `man` or `burn`:

```
new Query(Person.class, "name ~~ \"*man\" || name ~~\"*burn\"");
```

- Matching a name using a single wildcard character with a bound variable:

```
FreeVariables vars = new FreeVariables();
vars.put("var", String.class);
Query query = new Query(Person.class,"name ~~ var", vars);
FreeVariableBindings bindings = new FreeVariableBindings();
bindings.put("var","*Gr?y");
query.select(coll, bindings);
```

- Matching a name using a case-insensitive match for `?foo`:

```
new Query(Person.class,"name ~~ \"&i&?foo\"");
```

- Matching a name using a case-insensitive match for `*foo` appearing anywhere:

```
new Query(Person.class,"name ~~ \"&i*&*foo*\"");
```

- Matching a name `foo` appearing anywhere followed by `&bar`:

```
new Query(Person.class,"name ~~ \"*foo*&&bar*\"");
```

- Matching the name `(a)`:

```
new Query(Person.class,"name ~~ \"&(a&)\"");
```

Using Free Variables in Queries

Free variables are lexically the same as identifiers in the Java language. If you use free variables in your query, you must specify them in an optional third argument to the `Query` constructor. Use the `com.odi.util.query.FreeVariables` class. This class implements the `Map` interface. In addition, it provides type checking to ensure that the keys and values are `String`s and `Class`es, respectively. For example:

```
FreeVariables vars = new FreeVariables();
vars.put("INPUT_SALARY", Integer.TYPE);
Query q = new Query(Person.class,
  "salary>=INPUT_SALARY", vars);
```

When you execute a query, you must bind any free variables to particular values. Do this by passing an additional argument to the `Query.select()` or `Query.pick()` method. This argument must be of type `com.odi.util.query.FreeVariableBindings`. This class, like `FreeVariables`, implements the `Map` interface and provides additional type checking to ensure that the keys are `String`s.

The values you bind to the free variables must be of the type specified by the corresponding entry in the `FreeVariables` map that was specified at query construction. For primitive types, the type of value stored in the `FreeVariableBindings` must be the associated wrapper type. PSE Pro does not check that the correct types are bound until it executes the query.

For example, the INPUT_SALARY free variable is used in the previous example query. Your application might read in a value from a user in an interactive program or compute the value in some other way. Regardless of how your application computes the value, the free variable is bound to a specific value only when the query is executed. For example:

```
int INPUT_SALARY = {user input or some other computation};
FreeVariableBindings bindings = new FreeVariableBindings();
bindings.put("INPUT_SALARY", new Integer(INPUT_SALARY));
Set result = q.select(employees, bindings);
```

Executing Queries

You can execute a query that

- Specifies predefined variables or free variables
- Returns one element or a set of elements

Obtaining a set

To obtain the set of elements that satisfy a query, call the `com.odi.util.query.Query.select()` method. The two overloadings follow:

```
public Set select(Collection coll)
```

```
public Set select(Collection coll,
  FreeVariableBindings freeVariableBindings)
```

The *`coll`* argument specifies the collection to be queried. If this query has been explicitly optimized with the `Query.optimize()` method, any indexes specified in the optimization must be available on this collection. If this query has not been explicitly optimized, PSE Pro optimizes it for all indexes on the collection being queried. If the query has been explicitly optimized for indexes that are not available on the specified collection, PSE Pro signals `QueryIndexMismatchException`.

The *`freeVariableBindings`* argument specifies a `FreeVariableBindings` object that defines bindings for each free variable in the query. For each entry, the key is a `String` that identifies the free variable, and the value is the value that should be associated with the free variable during the evaluation of the query. The value must be of the type specified by the corresponding entry in the *`FreeVariable`* argument passed to the `Query` constructor. For the query to be evaluated, every free variable associated with the query when it was constructed must have a corresponding binding. Also, every free variable binding must correspond to a free variable that was specified when the query was constructed. If the free variable bindings do not match the free variable definitions specified when the query was constructed, PSE Pro signals `QueryException`.

The `select()` method returns a newly allocated transient `Set` that contains the elements that satisfy the query. If PSE Pro does not find any matching elements, it returns an empty collection. The returned `Set` is transient.

Obtaining a single element

To obtain one element that satisfies a query, call the `com.odi.util.query.Query.pick()` method. Following are the two overloadings:

```
public Object pick(Collection coll)

public Object pick(Collection coll,
  FreeVariableBindings freeVariableBindings)
```

The *`coll`* and *`freeVariableBindings`* arguments are the same as for the `select()` method. The `pick()` methods return the first element found that satisfies the query. If no elements in the collection satisfy the query, PSE Pro signals `NoSuchElementException`.

Type of returned element

The `select()` and `pick()` methods never return elements that are not of the class that was specified as the collection element type when the query was constructed.

Null values

Queries ignore null elements but not null fields. The result set of a query never includes null elements. When a query reaches a null element, execution continues to the next element. Suppose you have a query like the following:

```
name != "fred"
```

A query that evaluates this on a collection returns elements with null `name` fields as well as elements with names that are not `"fred"`.

Now suppose you have a query like the following:

```
spouse.name != "fred"
```

On a collection that includes elements that do not have spouses, this query does not return those elements without spouses. It returns only the elements that have spouses with names that are not `"fred"`, plus the elements that have spouses with null `name` fields.

Limitations on Queries

When a query refers to a class or field, the class or field must be public.

When a query refers to a method, the method must return something. In other words, in a query string, you cannot refer to a method that returns void.

Enhancing Query Performance with Indexes

When you want to run a query on a particularly large collection, it is useful to build indexes on the collection to accelerate query processing. An index provides a reverse mapping from a field value, or from the value returned by a method when it is called, to all elements that have the value. A query that refers to an indexed member executes faster because it is not necessary to examine each object in the collection to determine the elements that match the predicate. Also, PSE Pro does not need to fetch into memory every element.

This section discusses the following topics:

- How Indexes Work
- Adding Indexes to Collections
- Dropping Indexes from Collections
- Using Multistep Indexes in Queries
- Sample Program That Uses Indexes
- Sample Program That Queries User-Defined Fields
- Modifying Index Values
- Managing Indexes and Index Values
- Optimizing Queries for Indexes
- Manipulating Indexes Outside the Query Facility

How Indexes Work

When you add an index to a collection, PSE Pro examines every element of the collection to determine the value of the indexed field or method. After you build the index, you can run queries against the collection without reexamining the elements to determine the values of any indexed members. The query examines the index instead of the collection.

A query can include both indexed fields and methods and nonindexed fields and methods. PSE Pro evaluates the indexed fields and methods first and establishes a preliminary result set. PSE Pro then applies the nonindexed fields and methods to the elements in the preliminary result set.

Adding Indexes to Collections

You can add indexes to any collection that implements the `com.odi.util.IndexedCollection` interface, directly or indirectly. Note that the `IndexedCollection` interface extends the `Collection` interface.

The `IndexedCollection` interface provides methods for adding and removing indexes and updating indexes when the indexed data changes. In this release of PSE Pro, `com.odi.util.OSTreeSet` is the only collection class that already implements `IndexedCollection`. You can, of course, define other `Collection` classes that implement `IndexedCollection`. Call the `com.odi.util.IndexedCollection.addIndex()` method to create an index. Following are the three overloadings:

- `addIndex(Class` *`elementType`*`, String` *`path`*`)`
- `addIndex(Class` *`elementType`*`, String` *`path`*`, boolean` *`ordered`*`, boolean` *`duplicates`*`)`
- `addIndex(Class` *`elementType`*`, String` *`path`*`, boolean` *`ordered`*`, boolean` *`duplicates`*`, Placement` *`placement`*`)`

The *`elementType`* argument indicates the type to which the index applies. Objects of other types can be in the collection that you index but they are ignored by the index. A query that uses the index does not return such elements.

The *`path`* argument indicates the member to be indexed. A method member can have no arguments or one constant argument. The path can be either the name of a public field or a call to a public instance method, where the public instance method can be in a superclass. The path can also designate a complex navigation path through multiple public data members, such as `a.b().c.name`. If the syntax is incorrect, PSE Pro signals `IndexException`.

The *`ordered`* and *`duplicates`* arguments allow you to specify whether the index is ordered and whether it allows duplicates. If you do not specify the `boolean` arguments, the index is unordered and it allows duplicates.

Finally, the `Placement` parameter indicates the database or segment in which to store the index. If you do not pass a `Placement` argument, PSE Pro stores the index in the same database, segment, and cluster as the collection.

Dropping Indexes from Collections

Call the `com.odi.util.IndexedCollection.dropIndex()` method to remove an index from a collection. Following is the method signature:

`public boolean dropIndex(Class` *`elementType`*`, String` *`path`*`)`

The *`elementType`* argument indicates the type to which the index applies.

The *`path`* argument indicates the member for which the index is being removed.

If the index being dropped is a primary index from an `OSTreeSet`, the method replaces the primary index with a map that uses object hash codes to find the objects in the `OSTreeSet`.

Note Removing an index from a collection destroys the index.

Using Multistep Indexes in Queries

You can create a query that uses a *multistep index*, which is an index on a complex navigational path that accesses multiple public data members. It optimizes queries that use that same path. For example, if you wanted to know all employees whose supervisor has a salary less than `50,000`, you could create a multistep index and use it in your query as follows:

```
public class Employee {
  public int salary;
  public Employee supervisor;
  ...
}

// Getting the employees collection
Collection employees = (Collection)db.getRoot("employees");

// Adding the multistep index
employees.addIndex(Employee.class, "supervisor.salary",
      true /*ordered*/, true /*duplicates*/);

  ...
//Query using the multistep index
Query q=new Query(Employee.class, "supervisor.salary < 50000");
Set result = q.select(employees);
```

Sample Program That Uses Indexes

In the com/odi/demo/query directory, the QueryCustomers class includes the following example of using an index:

```
IndexedCollection collection = new OSTreeSet(db);
try {
  collection.addIndex(Employee.class, "salary");
}   catch (IllegalAccessException e) {
  System.err.println("Couldn't access field: " + e);
  System.exit(1);
}
Set result = q.select(employees);
```

Sample Program That Queries User-Defined Fields

In the com/odi/demo/props directory, the generic PropertiesObject class allows you to create instances in a database without defining the Java classes for the schema. For more information, see the README file.

Modifying Index Values

After you add an index to a collection, PSE Pro maintains it automatically as you add or remove elements from the collection. However, it is your responsibility to manage index maintenance when indexed members are modified for instances that are already members of an indexed collection.

For example, suppose you insert Lee into your collection of employees. You build an index for this collection on the phoneExtension field. A query of "phoneExtension == 1234" returns Lee. If you remove Lee from the collection, PSE Pro updates the index so it no longer includes Lee. However, if you leave Lee in the collection but change Lee's phone extension, you must manually correct the index so that Lee refers to the correct phone extension.

Methods

There are three methods that you can use to manually maintain an index:

- IndexedCollection.removeFromIndex() removes a value from the index.
- IndexedCollection.addToIndex() inserts a value into the index.
- IndexedCollection.updateIndex() removes a value from the index and replaces it with a value that you specify.

After an application calls one of these methods, the next time the application uses that index, it uses the updated index. A call to updateIndex() does the same thing as a call to removeIndex() followed by a call to addToIndex(). The exception is that removeIndex() and addToIndex() inspect the value to determine the index key. That is, they apply the index's path expression to obtain the key from the value. With updateIndex(), you pass in the old key and the new key. PSE Pro does not have to inspect the value to determine its key. For this reason, and because there is a single call, using updateIndex() is more efficient.

Removing and adding index values

The `removeFromIndex()` method has the following two overloadings:

```
public void removeFromIndex(Object value)

public void removeFromIndex(Class elementType,
  String path, Object value)
```

The `addToIndex()` method has the following two parallel overloadings:

```
public void addToIndex(Object value)

public void addToIndex(Class elementType,
  String path, Object value)
```

Usually, after you remove a value from an index, you should add a value to replace it.

If you know exactly the value that you need to add or remove, you can use the form that specifies *elementType*, *path*, and *value*. If you do not know the indexes that exist, or if you modified a lot of different fields and want to update all indexes, use the short form. In this case, PSE Pro iterates over all indexes and updates all of them.

Following is an example of removing and adding values to an index:

```
Employee lee = new Employee("Lee", 1234);
collection.insert(lee);
try {
  collection.removeFromIndex(lee);
  lee.setExtension(5678);
  collection.addToIndex(lee);
}   catch (IllegalAccessException e) {
  System.err.println("Could not access field: " + e);
  System.exit(1);
}
```

Updating indexes

The `updateIndex()` method has the following signature:

```
public void updateIndex(Class elementType,
  String path, Object oldKey, Object newKey, Object value)
```

Following is an example of updating an index:

```
Employee lee = new Employee("Lee", 1234);
collection.insert(lee);
lee.setExtension(5678);
collection.updateIndex(
  Employee.class, "extension",
  new Integer(1234), new Integer(5678), lee);
```

Managing Indexes and Index Values

When you add or drop an index, you do it at the class level. That is, you specify the class and member that the index is on. For example, you might add an index on the `name` field of the `Employee` class, as follows:

```
employeeCollection.addIndex(Employee, "name")
```

However, when you perform maintenance on an index, that is, when you call `removeFromIndex()`, `addToIndex()`, or `updateIndex()`, you do it at the instance level. For example, suppose you have an employee named Jones with an employee ID number of 1234. The employee's name changes to Smith. You must update this index entry at the instance level by

1. Removing the object from the index while the object still has its old key value
2. Adding the object back into the index with the new key value

The following example shows how to update the index when an employee's name changes from Jones to Smith:

```
employeeCollection.removeFromIndex(employee1234);
employee1234.setName("Smith");
employeeCollection.addToIndex(employee1234);
```

For each index on the `Employee` class, these methods update the index's value for `employee1234`. If there are multiple indexes on `Employee`, the one-argument overloading of `removeFromIndex()` and `addToIndex()` updates all of them. You need not specify that you want to update the index on the `name` field. For example, there might be indexes on the `Employee.salary` and `Employee.location` fields, as well as the `Employee.name` field. The previous code fragment would update the indexes on `salary` and `location`, as well as the index on `name`, even though only the index on `name` needs to be updated. This technique is useful when you make a lot of changes to different fields.

If you use the three-argument overloading of `removeFromIndex()` or `addToIndex()`, you can update only the index that needs to be updated. You must know the type of the indexed element, the name of the indexed member, and the value to be removed or added. For example:

```
employeeCollection.removeFromIndex(
  Employee.class, "name", employee1234);
employee1234.setName("Smith");
employeeCollection.addToIndex(
  Employee, "name", employee1234);
```

You can also update an index by using the `updateIndex` method. Instead of removing the object with its old key value from the index and then adding the object with its new key value back into the index, you can just call the `updateIndex` method.

If you call the `updateIndex` method by using the previous example, and then your code would look like the following:

```
employeeCollection.updateIndex(
  Employee.class, "name", employee1234.name, "Smith", employee1234)
employee1234.setName("Smith");
```

Optimizing Queries for Indexes

If you do not explicitly optimize a query for a particular set of indexes, PSE Pro optimizes the query automatically when it applies the query to a collection. This means that PSE Pro optimizes the query to use exactly those indexes that are available on the collection being queried.

Preparation

Before you optimize a query, you must obtain an instance of `IndexDescriptorSet`. An `IndexDescriptorSet` implements a set of `IndexDescriptor` objects. An `IndexDescriptor` is an object that describes an `IndexMap` on an instance of `IndexedCollection`. Typically, you can obtain an `IndexDescriptorSet` with a call to `IndexedCollection.getIndexes()` on any collection that has exactly the indexes for which you want to optimize your query.

Explicit optimization

To explicitly optimize a query, call the `Query.optimize()` method. The method signature is

```
public synchronized void optimize(IndexDescriptorSet indexes)
```

The *`indexes`* argument is an instance of `IndexDescriptorSet` that contains `IndexDescriptor` objects that describe the indexes against which to optimize.

Reoptimizing

If you apply an optimized query to the same collection again or to another collection with the same indexes, PSE Pro uses the same optimization. Reoptimization is not required. However, suppose you apply an optimized query to a collection that does not have all the indexes that were present when the query was first run. In this situation, PSE Pro must reoptimize the query. PSE Pro does this automatically; your intervention is not required.

Manual optimization

Automatic index optimization is convenient and effective. However, suppose a query is to be run multiple times against more than one collection, potentially with different indexes available. In this situation, it might be best to manually control the query optimization strategy.

For example, consider that the same query is to be run repeatedly against two different collections, in which the collections have different indexes. One alternative is to create two separate query objects, one for each collection. This avoids the overhead of recomputing the indexing optimization strategy each time you apply the query to a different collection. A second alternative is to explicitly optimize a query to use only the intersection of the indexes that are available on both collections. You can do this with a call to `Query.optimize()`. Pass in an `IndexDescriptorSet` object that contains descriptions of only the common indexes.

Restriction

If you explicitly optimize a query with the `Query.optimize()` method, it cannot run against a collection that does not have the specified indexes. If you try to do this, PSE Pro signals `QueryIndexMismatchException`. In this way, an explicitly optimized query differs from an automatically optimized query. An automatically optimized query reoptimizes itself as needed when you run it against a collection with different indexes.

This might be useful when it would be undesirable to run a particular query on a collection that does not have the required indexes. For example, this is useful when

the collection is very large and the overhead of examining every element of the collection is prohibitive.

To evaluate query expressions efficiently, PSE Pro compiles query expressions into classes and methods that are loaded when the query is evaluated. Each new query potentially can result in the creation of a new class with a new internal name to represent the compiled state of the query. When the query is no longer referenced, this class is normally garbage collected by the Java VM GC and its storage reclaimed.

-Xnoclassgc option to Java VM

If you are using Java collection classes with the `-Xnoclassgc` option to the Java VM, you risk running out of heap storage as the query expression classes that are generated by PSE Pro accumulate over time. The `-Xnoclassgc` option prevents the query expression classes from being garbage collected.

Manipulating Indexes Outside the Query Facility

You can use the IndexMap interface to directly access and manipulate indexes outside the query facility. This interface is useful when you want a sorted result set and you can represent the query as a single range expression on an indexed member. Instead of running a query, you can iterate over the index directly. See the on-line *Java API Reference* for more information on `com.odi.util.IndexMap`.

Storing Objects as Keys in Persistent Hash Tables

The `com.odi.util.OSHashtable` class introduces a new requirement for classes of objects that will be stored as keys in persistent collections: these classes must provide a suitable `hashCode()` method. PSE Pro and the class file postprocessor provide facilities for doing this conveniently.

This section discusses the following topics:

- Requirements for Hash Code Methods
- Providing an Appropriate Persistent Hash Code Method
- Storing Built-In Types as Keys in Persistent Hash Tables

Requirements for Hash Code Methods

Objects that are stored as keys in persistent hash tables must provide hash codes that remain the same across transactions. PSE Pro can create a new transient Java object in each transaction to represent a particular persistent object, so it is important that the `hashCode()` method used for persistent objects return the same hash code for these different transient objects.

The default `Object.hashCode()` method supplies an identity-based hash code. This identity hash code might depend on the virtual memory address or some internal implementation-level metadata associated with the object. Such a hash code is unsuitable for use in a persistent identity-based hash table because it would effectively be different each time an object was fetched in a transaction.

Providing an Appropriate Persistent Hash Code Method

In cases in which a persistence-capable class does not override the `hashCode()` method it inherits from `Object`, the class file postprocessor arranges for the class to implement a `hashCode()` method suitable for storing instances in persistent hash tables. It does this by adding an `int` field to the class. This field is initialized to an appropriate hash code when an instance is created and returns the value stored in the field from its `hashCode()` method. This hash code value is guaranteed to remain unchanged for the lifetime of the object.

Applications need to provide their own `hashCode()` methods for classes that define `equals()` methods that depend on the contents of instances rather than on object identity. If the `equals()` method only uses the == operator to compare the argument with `this` (or inherits `Object.equals()`), it is identity based and the `hashCode()` method provided by the class file postprocessor is appropriate. If the `equals()` method compares the contents of the objects, it is contents based and your application must supply a `hashCode()` method that returns the same hash code value for all objects whose contents make them return `true` when compared to the `equals()` method.

If an application does not need to store instances of a particular persistence-capable class as keys in a persistent hash table, there is no special requirement for that class's `hashCode()` method. In this case, to avoid making all your instances one word larger, have the class define or inherit a `hashCode()` method that calls the superclass's `hashCode()` method:

```
public int hashCode() {return super.hashCode();}
```

Doing this ensures that the `hashCode()` method inherited from `Object` is used, which returns a hash code that can be used only in a nonpersistent context.

Storing Built-In Types as Keys in Persistent Hash Tables

You can use the following built-in Java types as `OSHashtable` keys without overriding the `hashCode()` method:

- `java.lang.String`
- Wrapper classes, for example, `Character`, `Integer`, `Long`, `Float`

There is no way to override the `hashCode()` method for arrays. Therefore, do not use Java arrays as keys in persistent hash tables. You can, however, define a class that stores the array as a field and provides an appropriate `hashCode()` method.

Java wrapper classes work well as keys because their `hashCode()` methods are based on the value of the object rather than on its address.

Using Third-Party Collections Libraries

You can use a third-party Java collections library with PSE Pro. The advantages of doing so are that it might have features that you need or that you might be familiar with its use. The disadvantage is that it might not scale to the degree that you need.

Chapter 8
Generating Persistence-Capable Classes Automatically

This chapter provides information and instructions for using the class file postprocessor to make classes persistence capable. Reference information for all postprocessor options is in Chapter 13, Tools Reference, on page 271. For information on what Java-supplied classes are persistence capable, see Java-Supplied Persistence-Capable Classes on page 261.

Caution

For simple applications, it is best to postprocess all classes together. For more complex applications, you can postprocess your classes in correctly grouped batches. See Postprocessing a Batch of Files Is Important on page 173.

Failure to postprocess the correct classes together can result in problem situations that appear when you try to run the application and that are difficult to diagnose. There are postprocessor options that allow you to determine those classes that are made persistence capable.

Contents

This chapter discusses the following topics:

Overview of the Class File Postprocessor

To store an object in a database, the object must be persistence capable. For an object to be persistence capable, it must include code that allows persistence. PSE Pro includes the class file postprocessor utility to insert the required code, referred to as *annotations*, into your class files automatically.

Annotating classes automatically

The command you use to run the class file postprocessor command-line utility is `osjcfp`. The postprocessor provides a number of command options that allow you to tailor the results to your needs.

You can run the postprocessor (or its companion API) on classes or class libraries that you create or that you purchase from a vendor. See `com/odi/demo/collections/README.htm` for an example of making a third-party library persistence capable.

Note

You must explicitly postprocess each class that you want to be persistence capable by using the `osjcfp` utility. When you extend a persistence-capable class, objects do not inherit persistence, which is the ability to be stored in a database.

When you postprocess or manually annotate a class, this registers the class with PSE Pro. If a class is not postprocessed or manually annotated, PSE Pro signals `ClassNotRegisteredException`.

This overview provides the following information:

- Description of the Annotations
- Description of the Process
- Postprocessing a Batch of Files Is Important
- Manual Annotation

Description of the Annotations

The class file postprocessor annotates classes you define so that they are persistence capable. This means that the postprocessor makes a copy of your class files, places them in a directory you specify (either the source directory or another directory), and adds byte-code instructions (annotations) that are required for persistence.

These annotations are

- Modifying the class to implement the `com.odi.IPersistent` interface.
- Defining methods to initialize instance fields with data from the database, writing modified fields to the database, and resetting instance fields to default values.
- Modifying methods to fetch the contents of persistent instances from the database as needed and to mark modified instances so their changes can be written to the database at transaction commit.

  Before an application can access the contents of a persistent object, it must call the `ObjectStore.fetch()` method to read the object or the `ObjectStore.dirty()` method to modify the object. These calls make the contents of the object available

to your application. The postprocessor inserts these calls in methods of classes that it makes persistence capable or persistence aware.

- Defining an additional class that provides schema information about the persistence-capable class, if specified or required. This new class is a subclass of the `com.odi.ClassInfo` class.

Description of the Process

Before you run the postprocessor, you must compile your source files. The set of files you run the postprocessor on can contain a combination of class files, `.zip` files, and `.jar` files. The postprocessor generates annotated class files and places them in a directory that you specify.

This destination directory is never the original directory unless you specify the `-inplace` option (see page 274). When you are in a development cycle, it is best to specify a directory other than the original directory. Doing so avoids errors and provides both a persistence-capable and a transient version of the same class.

It is not necessary to recompile all classes before iteratively running the postprocessor. The requirement is that the compiled classes be consistent.

The postprocessor tries to minimize the amount of work it does. It checks file modification times and reprocesses only those files that have changed.

Postprocessing a Batch of Files Is Important

In one execution of the postprocessor, the postprocessor must operate on a correctly grouped set of files. For example, an application might use a file, perhaps a library, that is already annotated. You must not specify the annotated files when you run the postprocessor on the rest of the files in your application. Hence, the term *batch* means all files that the postprocessor must annotate in one execution of the `osjcfp` command. Each batch must have its own postprocessor destination directory for this to work correctly.

You can use the postprocessor `-inplace` option to create multiple batches. When you do, there is no requirement for the separate batches to be stored in different directories.

Example of one batch

When you write a program that uses persistence, the program usually consists of a batch (a set) of classes, for example, classes `A`, `B`, and `C`. They typically are defined in files called `A.java`, `B.java`, and `C.java`. It is possible for each class to reference the other classes. For example, `B` might refer to `C`, and `C` might refer to `B`. There is no ordering or layering; there are no rules for references among the classes.

When this is the scenario, you must run the postprocessor on all of these classes at the same time. You cannot run the postprocessor on each file individually. This is because when the postprocessor operates on `A`, it might refer to `B` and `C`. The postprocessor must have information about `B` and `C` to correctly annotate `A`.

Example of two batches

In relatively simple programs, there is only one batch involved. However, sometimes there might be more than one batch in an application. Suppose, for example, that you want to write a persistent program that uses an existing library.

An example of this is `djgl`, which is the persistence-capable version of ObjectSpace's JGL library. Your program consists of `A`, `B`, and `C` plus the JGL library.

In a simple (one-batch) program, when you run the postprocessor, you always specify all files in your application. In this case, you do not want the postprocessor to operate on JGL because it has already been postprocessed. In fact, you probably do not have the class files that have not been postprocessed.

It is correct to run the postprocessor on only `A`, `B`, and `C`. This is because there is a rule: JGL classes never know about `A`, `B`, and `C`. After all, JGL was written, finished, and put on the shelf before `A`, `B`, and `C` were created.

There are two batches here:

- The first batch contains the persistence-capable JGL library. runs the postprocessor on this batch.
- The second batch contains your own classes, `A`, `B`, and `C`. You run the postprocessor on this batch.

Whenever you run the postprocessor, you must run it on a whole batch. Each batch must have its own postprocessor destination directory.

Checking for correct batches

To determine whether you have correctly grouped your files in batches, you can apply this rule: Class `A` and class `B` must be in the same batch if either of the following is true:

- Class `B` inherits from class `A` and either class is persistence capable.
- Class `A` is persistence capable or persistence aware and it refers directly to the fields of class `B`, which is persistence capable.

Postprocessor API

PSE Pro also includes a companion API for it class file postprocessor utility for those times when it is useful to call the postprocessor from your Java application. The method that you call is

`com.odi.filter.OSCFP.filter(String[] argv)`

For information on how to use this method, see the Postprocessor API on page 279.

Manual Annotation

In exceptional situations, you might want to insert all required annotations needed for persistence and not use the postprocessor at all. For more information, see Chapter 9, Generating Persistence-Capable Classes Manually, on page 207. You can also manually annotate your code to meet some persistence requirements and then run the postprocessor to insert the other annotations.

Running the Postprocessor

To make classes persistence capable, do the following:

1 Compile the source files.

2 Run the postprocessor on the resulting class files.

You must run the postprocessor on all class files in a batch at the same time.

Some Java-supplied classes are persistence capable. Others are not persistence capable and cannot be made persistence capable. A third category of classes can be made persistence capable but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 261.

The topics discussed in this section are

- Preparing to Run the Postprocessor
- Requirements for Running the Postprocessor
- Example of Running the Postprocessor
- About the Postprocessor Destination Directory
- How the Postprocessor Interprets File Names
- Order of Processing
- How the Postprocessor Handles Duplicate File Specifications
- How the Postprocessor Handles Files Not Found
- .Zip and .Jar Files as Input to the Postprocessor
- How the Postprocessor Handles Previously Annotated Classes
- Troubleshooting OutOfMemory Error
- How the Postprocessor Handles Inner Classes
- When ClassInfo.java Files Are Generated

Preparing to Run the Postprocessor

Before you run the postprocessor, ensure that the following `.jar` files are explicitly specified in your `CLASSPATH`. An entry for the directory containing them is not sufficient.

- A `tools.jar` entry must be in your `CLASSPATH` environment variable.
- If you are using PSE Pro, the `pro.jar` file must be in your `CLASSPATH` environment variable.
- Also, you must update your `PATH` environment variable to contain the `bin` directory from the PSE Pro for Java distribution.

  On Windows, you might set `PATH` to be something like this:

```
PATH=c:\winnt\system32;
  c:\winnt;
  c:\jdk15\bin;
  c:\ODI\PSEProJ\bin
```

On UNIX, it would be something like this:

```
PATH=/home/mydir:
  /usr/local/odi/pseproj/bin:
  /usr/local/jdk15/bin
```

Requirements for Running the Postprocessor

The postprocessor requires specification of

- The `-dest` option with a destination directory for the annotated class files. This can be an absolute or a relative path name. This directory must already exist when you specify it on the postprocessor command line. The postprocessor does not create it.

  You can also specify the `-inplace` option to instruct the postprocessor to overwrite your original class files. If you do, the `-dest` option is still required.

- A batch of files. A batch includes all files in your application except already annotated files that your application refers to. You can specify
  - One or more `.class` files
  - One or more `.zip` files
  - One or more `.jar` files
  - One or more class names (you must specify the package names)
  - Any combination of the previous items

Insert a space between specifications and be sure to specify the required destination parameter. When you run the postprocessor, each batch must have its own destination directory. For example:

```
osjcfp -dest osjcfpout com.odi.demo.threads.Institution Banking.jar Account.class
```

You can specify additional options, which are described in osjcfp: Running the Postprocessor on page 272.

Example of Running the Postprocessor

To make the `Person` class persistence capable, enter a command such as the following:

```
osjcfp -dest ..\osjcfpout Person.class
```

This command assumes that the `Person.class` file is in the current directory and the `osjcfpout` directory is a sibling to the current directory. When the postprocessor successfully generates an annotated version of the `Person.class`, the file contains the information PSE Pro needs to store instances of `Person` persistently.

The postprocessor places the annotated `Person.class` file in a package-relative subdirectory of the `osjcfpout` directory. For example, suppose the `Person` class package name is `com.odi.demo.people`. Further suppose that the `osjcfpout` directory is in the `\users\kim` directory. The postprocessor writes the annotated class file to a file whose name is made up of the destination directory, the class package, the class name, and the `.class` extension:

```
\users\kim\osjcfpout\comcom\odi\demo\people\Person.class
```

Note that both of the following commands have the same results, as specified previously:

```
osjcfp -dest ..\osjcfpout Person.class
osjcfp -dest \users\kim\osjcfpout com.odi.demo.people.Person
```

About the Postprocessor Destination Directory

The postprocessor never overwrites the class files specified on the postprocessor command line unless you specify the -inplace option when you run the postprocessor. If you do specify the `-inplace` option, the postprocessor overwrites the original class files with the annotated class files.

If you specify a destination directory in such a way that it would store the annotated class file in the same location as the unannotated class file and you do not specify the `-inplace` option, the postprocessor displays an error message and terminates. It does not produce any class file output.

The postprocessor ignores classes that are rooted in the destination directory. If you try to postprocess a class that exists only in the destination directory and you do not specify `-inplace`, the postprocessor reports that it cannot find the file. For example, if you specify the following command when you run `osjcfp`, you receive an error as shown:

```
setenv CLASSPATH
  /usr/devo/java/test:
  /opt/ODI/pseproj/lib/pro.jar:
  /opt/ODI/pseproj/lib/tools.jar
cd /usr/devo/java/test
javac com/users/jobs/teacher.java
osjcfp -d . com.users.jobs.teacher

Error: Class com/users/jobs/teacher could not be found.
```

Because the postprocessor ignores the destination directory in the `CLASSPATH` when it looks up classes, it is unable to locate the specified class. Consequently, the destination directory you specify cannot be the root directory for any of the classes you want to postprocess or any classes referenced by classes you want to postprocess.

Typically, after you run the postprocessor, you have a transient version of a class (your original file) and a persistence-capable version of the class (in the destination directory).

If there are no errors, the postprocessor places a version of all files specified on the command line in the destination directory. The postprocessor annotates those files that require annotations and does not modify those files that do not.

How the Postprocessor Interprets File Names

If a name you specify ends with `.class`, `.zip`, or `.jar`, the postprocessor assumes that it is an explicit file name for a class file, `.zip` file, or `.jar` file, respectively.

If a name you specify does not end with `.class`, `.zip`, or `.jar`, the postprocessor assumes that it is a class name delimited with periods, for example, `a.b.C`. The postprocessor uses the `CLASSPATH` environment variable or the `-classpath` specification on the postprocessor command line to locate the `.class` file, which can be in a `.zip` file or `.jar` file. (The use of the `-classpath` option does not affect the class path used for the execution of the postprocessor.)

`-classpath` example

Here is an example of adding the `-classpath` option. The command is entered all on one line.

```
osjcfp -dest osjcfpout
  -classpath /usr/local/odi/pseproj:
  /usr/local/odi/pseproj/lib/pro.jar
  com.odi.demo.threads.Institution Banking.zip Account.class
```

CLASSPATH and -classpath

The postprocessor uses the class path you specify in the command line to locate the specified files. This is in place of the `CLASSPATH` environment variable. At run time, Java implementations append the location of the system classes to the end of the `CLASSPATH` environment variable. You must do this manually if you specify the `-classpath` option. This is shown in the previous examples as `classes.jar`.

Order of Processing

The postprocessor processes the class files in the order in which they appear on the command line and according to the persistence mode that is in effect when the postprocessor reaches the file name. The persistence mode indicates whether the postprocessor is

- Annotating the class to be persistence capable
- Annotating the class to be persistence aware
- Copying the class to the destination directory without annotating it

Persistence mode options

The default persistence mode is that the postprocessor generates persistence-capable classes. Following are the options you can specify to determine the persistence mode:

Persistence Mode Option	*Description*
`-pc |-persistcapable`	Classes specified after this option are made persistence capable. The postprocessor annotates these classes to include all annotations required by PSE Pro for an object to be persistent. This is the default. If you do not specify a persistence mode option in the postprocessor command line, the postprocessor makes all specified classes and superclasses of those classes persistence capable.
`-pa |-persistaware`	Classes specified after this option are made persistence aware. The postprocessor annotates these classes so that they can operate on persistent objects, but instances of these classes cannot themselves be stored persistently.
`-cc |-copyclass`	Classes specified after this option are not annotated. Specify this option for classes that should not be annotated either because they are nonpersistent or are already annotated. The postprocessor copies these classes to the destination directory along with the annotated classes.

If you specify a `.class` file or class name, the postprocessor processes it according to the persistence mode that is in effect when the postprocessor reaches the file name. If you specify a `.zip` file or `.jar` file, the postprocessor processes all class files in the `.zip` file or `.jar` file according to the persistence mode that is in effect when the postprocessor reaches the name of the `.zip` file or `.jar` file in the command line. For example:

```
osjcfp -dest osjcfpout -persistaware Tent.class Family.class \
-persistcapable Campers.jar Site.class -copyclass Weather.class
```

Example

After you run the postprocessor with the previous command,

- The `Tent` and `Family` classes are persistence aware.
- The `Site` class, its superclass if it has one other than `java.lang.Object`, all classes in the `Campers.jar` file, and any of their superclasses (other than `java.lang.Object`) are persistence capable.
- The `Weather` class is not annotated and is copied as it is to the destination directory.

How the Postprocessor Handles Duplicate File Specifications

It is permissible for a class to be specified more than once in a command line. For example, a file can be in a `.zip` file and you can also explicitly specify it. On UNIX and on Windows NT using a Sun JDK, a file can be included in a wildcard specification and you can also explicitly specify it. In the previous example, the `Family` class could be in the `Campers.jar` file. If it were, the postprocessor would annotate the `Family` class to be persistence capable. This is because making a class persistence capable supersedes making it persistence aware. Likewise, making a class persistence aware supersedes copying it as is to the destination directory.

If you specify the same class more than once on a command line, both specifications must resolve to the same disk location. For example, suppose you specify both `Person.class` and `com.odi.demo.people.Person`. This is allowed only if the class path causes `com.odi.demo.people.Person` to resolve to the same `Person.class` that is explicitly specified.

How the Postprocessor Handles Files Not Found

The postprocessor must be able to find every file that you specify on the command line. If it cannot find one or more files, it displays an error message and stops processing. It does not produce any annotated class files.

.Zip and .Jar Files as Input to the Postprocessor

If a class originates in a `.zip` file or `.jar` file, either because you specify a `.zip` file or `.jar` file when you run the postprocessor or because the class path search locates the class in a `.zip` file or `.jar` file, the postprocessor writes the annotated class to the package-appropriate subdirectory of the destination directory.

How the Postprocessor Handles Previously Annotated Classes

If the postprocessor previously annotated a `.class` file, you can specify only that `.class` file to be copied. You cannot specify it to be annotated. If you do, the postprocessor displays a message that states the specified class that was already annotated and terminates without producing any annotated files.

Troubleshooting OutOfMemory Error

Java imposes a memory limitation of 16 MB unless you override it. If you receive a `java.lang.OutOfMemory` error during postprocessing, you must increase the run-time memory pool. Do one of the following:

- Set the `OSJCFPJAVA` environment variable to include the `-Xmx` option. For example, Solaris `csh` users can enter

  ```
  setenv OSJCFPJAVA "java -Xmx32m"
  ```

  Windows users can enter

  ```
  set OSJCFPJAVA=java -Xmx32m
  ```

- Edit the `osjcfp` script (Solaris) or `osjcfp.bat` script (Windows) to incorporate the `-Xmx` option in the invocation of Java near the end of the script. On Solaris, the line to change is

  ```
  $OSJCFPJAVA $javaargs com.odi.filter.OSCFP $args
  ```

  On Windows, the line to change is

  ```
  %osjcfpjava% com.odi.filter.OSCFP %1 %2 %3 %4 %5 %6 %7 %8
  ```

  Add `-Xmx32m` before the `com.odi.filter.OSCFP` entry. This allows the Java virtual machine to increase the heap to 32 MB. You can increase this value further if you need to.

How the Postprocessor Handles Inner Classes

When you define a class inside another class, you must explicitly make both the outer class and the inner class persistence-capable. For example, suppose you define the following class:

```
Class Foo {
  int a;
  public class Bar {}
}
```

You must specify both the `Foo` class and the `Bar` class when you run the postprocessor:

```
osjcfp -dest ../osjcfpout -pc Foo.class -pc Foo$Bar.class
```

When ClassInfo.java Files Are Generated

The postprocessor always generates *xxx*`ClassInfo` classes for persistence-capable classes that are private. However, by default, the postprocessor does not generate *xxx*`ClassInfo` classes for persistence-capable classes that are public. Instead, the postprocessor relies on the ability of PSE Pro to build an *xxx*`ClassInfo` class dynamically when needed, using the reflection API. This optimization reduces the disk footprint and application start-up times because there are fewer classes to load when the application starts.

The reflection API is subject to security and access constraints that are enforced to varying degrees at run time, depending on the version of your Java Virtual Machine and your platform. If your application encounters run-time security errors while attempting to generate *xxx*`ClassInfo` classes dynamically, specify the `-nooptimizeclassinfo` option when you run the postprocessor. When you specify `-nooptimizeclassinfo`, the postprocessor generates the *xxx*`ClassInfo` classes; therefore, the reflection API is not used.

Managing Annotated Class Files

After you run the postprocessor, there are two versions of your class files:

- The unannotated class files in the original directory
- The annotated class files in the destination directory

It is important to keep these versions separate because

- When you compile source code, you must ensure that any class files the compiler reads in are unannotated class files. The compiler must find unannotated class files before it finds annotated class files with the same names.

  Note: While the above is true for simple applications, it might not be true for more complex applications. See Using the Right Class Files in Complex Applications on page 184.

- When you run your application, you must ensure that PSE Pro finds the annotated class files before it finds the unannotated class files with the same names.

There are several ways to accomplish this. recommends that you

- Specify the `-classpath` argument to the compiler so it finds the unannotated class files first
- Modify your `CLASSPATH` environment variable so Java can find the annotated class files first when it runs your application

To help you manage annotated class files, this section discusses

- Ensuring That the Compiler Finds Unannotated Class Files
- Ensuring That PSE Pro Finds Annotated Class Files
- Using the Right Class Files in Complex Applications
- Alternatives for Finding the Right Files
- How the Postprocessor Determines Whether to Generate an Annotated Class File

Ensuring That the Compiler Finds Unannotated Class Files

The compiler can locate class files in the following two ways:

- Through the `CLASSPATH` environment variable
- Through the `-classpath` argument to the compiler

`CLASSPATH` is convenient when you compile, but when you try to run your application, PSE Pro finds the unannotated files before it finds the annotated files. The `-classpath` option is more cumbersome to use because it means that the path to Java system classes must be listed explicitly in the argument, but it is safe. It ensures that the compiler does not operate on annotated class files.

Example 1

For example, suppose that PSE Pro is installed in `c:\pse` and you are building an application in `c:\app`. Your destination directory for annotated class files is `c:\app\osjcfpout`. Your `CLASSPATH` variable might look like the following:

```
CLASSPATH=c:\ODI\PSEProJ\lib\pro.jar;c:\app\osjcfpout;c:\app
```

When you run the compiler, specify the `-classpath` option with the following path. This removes the destination directory from the class look-up path and adds the Java classes to the path.

```
javac -classpath c:\app;
  c:\ODI\PSEProJ\lib\pro.jar;
  c:\jdk1.2\lib\classes.jar App.java
```

Example 2

Following is an example of why it is important for the compiler to operate on unannotated class files. Suppose you have two classes named `X` and `Y` in the same postprocessor batch. Neither of these classes is explicitly declared to implement `com.odi.IPersistent`. Now suppose you add the following two methods to class `Y`:

```
void foo(com.odi.IPersistent p) {}
void bar() { foo(new X()); } // Trying to pass an X instance to
      // a function that is expecting com.odi.IPersistent
```

If you recompile only `Y.java` and the compiler finds the annotated classes, examination of the annotated class file allows the compiler to determine that `X` implements `IPersistent`, which allows `Y.bar()` to compile. If you then recompile both `X` and `Y`, the compiler recognizes that `X` is not declared to implement `com.odi.IPersistent` and refuses to compile class `Y`, even though it compiled successfully earlier.

Ensuring That PSE Pro Finds Annotated Class Files

When you run your application, PSE Pro must find the annotated class files before it finds the unannotated class files. The recommended way to ensure this is to define a `CLASSPATH` environment variable that has the postprocessor destination directory before the source file directory.

Example

Consider the following example:

- You are building an application in `c:\app`.
- You create a directory named `c:\app\osjcfpout` to hold the annotated class files.

In this scenario, use the following `CLASSPATH`:

```
c:\app\osjcfpout;c:\app;c:\ODI\PSEProJ\lib\pro.jar
```

After you modify your `CLASSPATH` environment variable, you can run the postprocessor with no special action. The postprocessor excludes the destination directory from the class path when it does class-path-based searches.

Using the Right Class Files in Complex Applications

There are situations in which you want the compiler to read in annotated class files. In these cases, the referenced classes are similar to an independent library on which you are building your application. The referenced classes form a batch, which is a group of class files that must be postprocessed together. The other files in your application form a second batch.

Independent library

For example, suppose this second batch is named `X`. Specify the `-classpath` option so that it points to the

- Unannotated class files for any classes in `X`
- Annotated class files for any classes that are in other batches and are referenced by classes in `X`

This is the most common multiple-batch scenario. Your application is in one batch and the other batches are existing reusable libraries. Each batch has its own postprocessor destination directory.

Classes referenced by other classes

Now suppose that you are not using an existing library. Your application itself contains a group of referenced classes (first batch), then another group of classes (second batch) that reference the first batch. The following instructions show how to build your application in stages:

1. Compile the source files and postprocess the class files in the first batch. (This is the batch of files that are referenced by other classes in the application.)
2. Compile the source files in the second batch. You might not want to compile all files in this batch at the same time. Specify the `-classpath` option to point to the annotated class files in the first batch and any unannotated class files in the second batch.
3. Run the postprocessor on the second batch. Specify a destination directory that is different from the destination directory that was specified when the postprocessor operated on the first batch. You can package the result of postprocessing the second batch in a `.zip` file or `.jar` file.

Alternatives for Finding the Right Files

In some circumstances, updating your `CLASSPATH` environment variable might be cumbersome or might not work well with your development environment. (This is true for the Symantec Cafe product.) In these cases, you can copy annotated files back to the building directory. However, if you do this, you must remove the annotated files before you recompile. This ensures that subsequent compilation and postprocessing operate on unannotated class files.

Two other alternatives are to

- Delete the contents of the destination directory before you recompile
- Specify the `-classpath` option when you run the postprocessor, just as you did for compilation

How the Postprocessor Determines Whether to Generate an Annotated Class File

When you run the postprocessor, it checks whether any annotated file it is going to create already exists. If an annotated file does not already exist, the postprocessor generates it. If an annotated file does exist, the postprocessor compares the date on the compiled input file with the date on the annotated output file. If the input file date is after the output file date, the postprocessor generates a new output file. If the input file date is before the output file date, the postprocessor does not generate a new file. It assumes that the annotated file that already exists is still valid.

This works well when you run the postprocessor repeatedly with the same command line. However, when you change input parameters to the postprocessor, it is a good idea to remove the previously annotated class files from the destination directory. The reason for this is that a comparison of dates might not cause a new annotated file to be generated when the specification of a new input parameter requires a new annotated file to be generated.

To force the postprocessor to overwrite existing annotated files, specify the `-f` or `-force` option when you run the postprocessor.

Creating Persistence-Aware Classes

If you know that a class will never need to be stored persistently, you can run the postprocessor to make the class persistence aware. A persistence-aware class can operate on persistent objects but cannot be persistent itself. For an example of how you might use persistence-aware classes, see `com/odi/demo/pport/README.htm`.

Persistence-aware annotations require less space than persistence-capable annotations require. The postprocessor adds calls to `ObjectStore.fetch()` and `ObjectStore.dirty()` only where they are needed to operate on persistent objects. When the postprocessor makes a class persistence aware, it does not annotate that class's superclass. You need only make a class persistence aware, instead of copying it as is, if

- The class accesses fields of a persistence-capable class instead of using methods to access the fields.
- The class accesses elements of persistent arrays.

You must make a class persistence aware (or persistence capable) when it includes methods that obtain arrays from persistent objects.

Specifying the Postprocessor Command Line

To create a persistence-aware class, specify the `-pa` or `-persistaware` option followed by the names of the classes that you want to be persistence aware. For example:

```
osjcfp -dest osjcfpout -persistaware Compute.class
```

The preceding command line annotates `Compute.class` so that it has calls to the `fetch()` and `dirty()` methods.

No Changes to Superclasses

Another reason to make a class persistence aware is that doing so does not require changing its superclasses. This is important for classes such as `java.lang.Thread`, whose superclass should not be modified. `java.lang.Thread` is inherently transient, so it makes no sense for it to become persistent because it is not useful when you take it out of the database. Typically, Java system classes are restricted from annotations by the postprocessor.

How the Postprocessor Works

This section describes postprocessor behavior relative to various components in your application. It is important to be familiar with the information here so that the postprocessor produces the results you expect. The topics covered in this section are

- Ensuring Consistent Class Files
- Modifications to Superclasses
- Effects on Inheritance
- Location of Annotated Class Files
- Postprocessor Errors and Warnings
- Handling of final Fields
- Handling of Static Fields
- Which Java Executable to Use
- Line-Number and Local-Variable Information
- Using a Debugger
- Handling of finalize() Methods
- Description of Postprocessor Optimizations

Ensuring Consistent Class Files

When you run the postprocessor on more than one class file at a time, all specified classes must be consistent. To ensure class consistency, compile all classes together. The postprocessor does not detect inconsistencies among files it operates on. For example, suppose you modify and recompile a class without also recompiling its subclasses. This can cause inconsistencies, which the postprocessor does not detect when it annotates the class files.

Modifications to Superclasses

When you run the postprocessor to make classes persistence capable, it generates annotated class files for the specified classes and for any superclasses that are in the same packages as the specified classes. PSE Pro requires annotations to superclasses for all classes that the postprocessor makes persistence capable. If a superclass is not in the same package as one of its subclasses that is being made persistence capable, you must explicitly specify the superclass on the postprocessor command line.

Effects on Inheritance

If a class that the postprocessor is annotating has no superclass other than `java.lang.Object`, the postprocessor annotates the class to implement the `com.odi.IPersistent` interface.

Implementation of the `IPersistent` interface is mandatory for objects that you want to be persistent. You must define classes so that if they inherit from another class, it is a class that can implement `IPersistent`.

Every class inherits from the `Object` class, which defines the `hashCode()` method and provides a default implementation. For a persistent object, this default implementation often returns a different value for the same persistent object (the object on the disk) at different times. This is because PSE Pro fetches the persistent object into different Java objects at different times, in different transactions or different invocations of Java.

This is not a problem if you never put the object into a persistent hash table or other structure that uses the `hashCode()` method to locate objects. If you do put them in hash tables or something similar, the hash table or other structure that relies on the `hashCode()` method might become corrupted when you bring the objects back from the database.

To resolve this problem, you can define your own `hashCode()` method and base it on the contents of the object so it returns the same thing every time. The signature of this method must be

```
public int hashCode()
```

If you do not provide a `hashcode()` method, the postprocessor adds one if it is necessary. If the default behavior of the postprocessor is not ideal for your application, you can specify the `-hashcode` and `-nohashcode` options to control where the postprocessor adds a `hashCode()` method.

Location of Annotated Class Files

When you run the postprocessor, you must specify a destination directory with the `-dest` option. The postprocessor uses the destination directory as the root directory of the class hierarchy of annotated files. The postprocessor places the annotated class file in the package-relative subdirectory of the destination directory. With the destination directory specified in your `CLASSPATH` environment variable, Java can find the annotated classes.

You must create the destination directory before you specify it in an `osjcfp` command line. The postprocessor creates the required subdirectories in the destination directory.

For example, suppose that you specify `osjcfpout` as the destination directory. When you run the postprocessor on the `Person.class` file, which is in the `com.odi.demo.people` package, the postprocessor places the annotated file in

```
osjcfpout\com\odi\demo\people\Person.class
```

The package name of the annotated class file remains the same, unless you specify an option to change it. The class name of the annotated class file is always the same as the class name of the unannotated class file.

Postprocessor Errors and Warnings

If an error occurs while the postprocessor is running, it terminates without writing any annotated class files.

For any warnings from the postprocessor, you might determine that you can safely ignore the warning. In this case, you can stop the postprocessor from warning you

about the field in question. To do so, specify the `-quietfield` option followed by the fully qualified name of the field for which you want to suppress warnings. Alternatively, you can specify `-quietclass` to suppress all warnings on the class.

Handling of final Fields

You cannot make `final` fields persistent. If you try to do this, the postprocessor displays a warning message and treats the fields marked as `final` as though you declared them to be `transient`. To allow such fields to be stored persistently, you must remove the `final` keyword.

Handling of Static Fields

The postprocessor never stores static fields in the database and never causes the values of static fields to be altered. You must write your own code to update static fields and to store static fields in the database, if that is what you want to do.

Static fields that can hold persistent values

The postprocessor displays a warning for a static field that can hold potentially persistent values. The postprocessor cannot determine the type of the object that is actually pointed to. Consequently, depending on the type of object referenced, the warning might not be applicable. For example, suppose you have a persistence-capable class named `X`. The `X` class has a static member named `y` of a type that implements `com.odi.IPersistent`. When you run the postprocessor, it displays a warning such as the following:

```
X.y is a static field of a type that implements com.odi.IPersistent
and that might refer to a persistent object. If this field does refer
to a persistent object it must be user maintained.
```

Referring to a persistent object

If the field mentioned in the warning is intended to refer to a persistent object, you can write your application as follows:

- When you create a database, create an object of the desired type and create a database root to refer to the object. This makes the object persistent and provides a mechanism that you can use to find the object.
- When you start a new transaction, if the object referenced by the static field is null or stale, look up the database root and set the value of the static field to the root value.

If you specify `ObjectStore.RETAIN_STALE` when you commit or abort a transaction, you must ensure that you correctly access the objects at the beginning of the next transaction. This is because PSE Pro does not make the object referenced by `X.y` persistent if it is only reachable from `X.y`. If PSE Pro makes it persistent because it is reachable from some other point, the object referenced by `X.y` might become stale at the end of the transaction in which it becomes persistent. If it does, and if the object referenced by `X.y` does become persistent, it is possible that the application might try to use the stale version of the object.

How can `X.y` be reachable from some other point? Perhaps another persistent object or an object that is going to be persistent refers to the object that the static data member is referring to. When PSE Pro commits the transaction and performs transitive persistence, it finds the object that the static data member is referring to.

References to stale objects

An issue to consider is stale references to stale objects. To avoid the inadvertent use of stale objects, update `X.y` at transaction boundaries. Set `X.y` to null or to another value to ensure that if a stale object is referenced by `X.y`, it is no longer accessible through `X.y`. Then you can suppress the warning with the `-quietfield` option.

Summary

For class `X`, the important points are listed below.

```
class X {static OSHashtable y = new OSHashtable();}
```

- `X.y` does not become persistent just because class `X` is persistence capable or because an instance of `X` becomes persistent.
- If you want `X.y` to become persistent, you must make it reachable from a root through a path that does not involve a static field, for example, `db.createRoot("X.y", X.y)`.
- If `X.y` does become persistent, you must be aware that the `OSHashtable` object referenced by `X.y` might become stale at transaction boundaries. If it does, you must update `X.y` to refer to a nonstale instance.

Which Java Executable to Use

The postprocessor is a Java program; it requires a Java virtual machine to run. It uses the first Java executable that it finds in your `PATH` environment variable. If you want the postprocessor to use another Java executable, set the `OSJCFPJAVA` environment variable to the name of the Java executable you want the postprocessor to use. The default is `java`.

If the postprocessor cannot find a Java executable, it generates a `Bad command or file name` error message.

Line-Number and Local-Variable Information

When the postprocessor annotates a class file, it maintains any existing line-number and local-variable information.

Using a Debugger

The class file postprocessor annotates methods with VM instructions for automatically performing `fetch()` and `dirty()` operations on objects. It does this in such a way that the debugging information in the class files remains intact. Typically, the annotations are invisible to an application. However, it is possible to encounter them under certain circumstances when using a debugger. For example, you might encounter the following when you use the `Step into` command:

```
x = foo(y.m);
```

Stepping into that statement might cause you to enter the PSE Pro code that causes the contents of the `y` object to be fetched. In such a situation, use the `Step out` command to leave the PSE Pro code. Then use the `Step into` command again, which should then step into the call to the `foo()` method.

You should rely on the `Step over` command whenever possible. However, there are situations in which you must use the `Step into` command. If you inadvertently step

into a PSE Pro method, step out of the PSE Pro code and return to your own code by doing one of the following:

- Use a `Step out` command.
- Set a breakpoint in the calling code.
- Use repeated `Step over` commands until the method returns.

Handling of finalize() Methods

The Java GC calls the `java.lang.Object.finalize()` method on an object that is no longer referenced. The GC does this before it frees the space occupied by the object. In this way, the `finalize()` method provides a hook that you can use to free resources that are not freed by garbage collection, for example, memory that was allocated by a native method call.

If your persistence-capable class defines a `finalize()` method (recommends that it should not), the class file postprocessor inserts annotations at the beginning of the `finalize()` method that change the persistent object to a transient object. See Avoiding finalize() Methods on page 131.

Description of Postprocessor Optimizations

The postprocessor optimizes `fetch()` and `dirty()` calls in several ways. If you determine that an optimization is preventing insertion of a required call to `fetch()` or `dirty()`, you can disable the optimization.

- For array objects in looping constructs, the postprocessor inserts the call to `fetch()` or `dirty()` only in the first loop iteration. To disable this optimization, specify the `-noarrayopt` option when you run the postprocessor. This causes the postprocessor to insert calls to `fetch()` or `dirty()` in every iteration.
- For access to fields relative to `this` in nonstatic member methods, the postprocessor optimizes calls to `fetch()` and `dirty()`. To disable this optimization, specify the `-nothisopt` option when you run the postprocessor. This causes the postprocessor to insert a `fetch()` or `dirty()` call for each access to a field in `this`.

You should disable these optimizations if you commit transactions or evict persistent objects as follows:

- If you call `commit()` or `evict()` while iterating over persistent array elements, specify `-noarrayopt` when you run the postprocessor.
- If you call `commit()` or `evict()` in between accesses to different fields of `this`, specify `-nothisopt` when you run the postprocessor.

Including Transient and Already Annotated Classes

After you run the postprocessor, the annotated class files are in the package-relative subdirectory of the destination directory (root directory) you specified. You might want other class files in this destination directory. These could be transient (nonpersistence-capable or nonpersistence-aware) class files or files that have already been annotated.

Copying Classes to the Destination Directory

To copy certain files to the destination directory along with the annotated files, specify the `-copyclass` option followed by the name of the file you want to copy. For example:

```
osjcfp -dest osjcfpout a.jar -copyclass b.class
```

In this example, the postprocessor annotates the files in `a.jar` and copies them to the package-relative subdirectory of the `osjcfpout` directory. The postprocessor also copies `b.class` to the `osjcfpout` directory but it does not modify the `b.class` file.

You can follow the `-copyclass` option with one or more `.class` file names, class names, `.jar` file names, or `.zip` file names. This option applies to each name that follows it until the postprocessor reaches a `-pc` or `-pa` option.

Specifying Classes to Be Copied and Classes to Be Persistence Capable

Classes for which you specify the `-copyclass` option can overlap with classes for which you specify the `-persistcapable` or `-persistaware` option. For example:

```
osjcfp -dest osjcfpout -copyclass *.class -persistcapable a.class
```

This allows you to keep all files in a package together and annotate only the classes that need to be annotated. You need not partition classes into those that need annotations and those that do not. You can specify the same file with more than one persistence-mode option because the `-persistcapable` option and the `-persistaware` option override the `-copyclass` option.

When Can a Class Be Transient?

Suppose you have a persistence-capable class, class `A`. A class that refers to class `A` can be transient if all access to `A`'s nontransient data members is through methods on `A`. The methods of `A` will be properly annotated. Because all other classes only use `A`'s methods, the other classes do not need to be persistence-aware. Consequently, you need not postprocess any classes that refer to `A`.

Any class that directly accesses `A`'s nontransient data members must be either persistence capable or persistence aware. Any other class that refers to `A` and does not directly access nontransient data members can be transient. That is, you do not have to postprocess it.

An important exception to this is that if a class manipulates an array object that might be persistent (specifically, setting and getting array elements), that class must be annotated to be persistence aware. However, if the code that provides access to the array is annotated to access the values of the array, you can avoid making the class persistence aware. It is difficult to reliably implement this in the general case.

If you compile with optimization the classes that use the methods that get and set array values, the compiler might inline the get and set methods. In this case, you must make the class that uses the get and set methods persistence aware.

Putting Processed Classes in a New Package

Normally, the postprocessor places the annotated files in a package-relative subdirectory of the destination directory, and the annotated files have the same package names as the original files. However, there is an option that allows you to change the package name of files specified in the postprocessor command line. The `-translatepackage` option modifies the package name so that the persistence-capable version of the class is in one package and the transient version (the original) is in another package.

To help you use the `-translatepackage` option, this section discusses the following topics:

- Using the -translatepackage Option
- How the Postprocessor Applies the Option
- Updating References to New Package Name
- References to Transient and Persistent Versions of a Class
- References to Transient Instances of a Persistence-Capable Class

Using the -translatepackage Option

To create persistence-capable classes whose package name is different from the original package name, specify the `-translatepackage` option followed by the current package name, then the new package name. The format for this option is

`{-translatepackage | -tp}` *`orig_pkg_name new_pkg_name`*

For example, suppose you have the `a.b.C` class and you want to create the `d.e.C` persistence-capable class. Run the postprocessor in the following way:

```
osjcfp -dest osjcfpout -translatepackage a.b d.e C.class
```

Exact match required

The specification for the original package name must exactly match the package name of the specified file. If there is not an exact match, the postprocessor does not place the annotated file in the new package. For example, suppose you have two classes named `com.odi.demo.New` and `com.odi.Old`. You want to move `com.odi.Old` to the `com.odi.beta` package and you specify the following command:

```
osjcfp -dest osjcfpout -tp com.odi com.odi.beta com.odi.demo.New com.odi.Old
```

The postprocessor places the annotated file for the `com.odi.Old` class in `com.odi.beta.Old` in the package-relative subdirectory of the `osjcfpout` directory (`osjcfpout\com\odi\beta\com.odi.beta.Old.class`).

The postprocessor does not place the annotated file for `com.odi.demo.New` in a different package because the original package name is `com.odi.demo` and not just `com.odi`. The postprocessor annotates `com.odi.demo.New` and places it in `osjcfpout\com\odi\demo\com.odi.demo.New.class`.

How the Postprocessor Applies the Option

The postprocessor applies the `-translatepackage` specification to

- All classes in the original package that are locatable by means of the `CLASSPATH` environment variable or the `-classpath` option, if you specify it. The `-classpath` specification overrides the `CLASSPATH` environment variable.
- Files on the command line whose package name exactly matches the specification for the original package name. This is true for files processed with the `-persistcapable`, `-persistaware`, or `-copyclass` option.

When copying files

It does not matter whether the postprocessor is making any other changes to the specified files. The postprocessor changes the package names of files for which the -copyclass option is specified along with new persistence-capable or persistence-aware files.

Multiple option specifications

You can specify this option more than once on a command line to specify several package translations. If you accidentally specify more than one translation for the same package, the postprocessor performs the last translation you specify in the command line.

Updating References to New Package Name

A change to the package name of a class requires updating all references to that class to reflect the new name.

The postprocessor updates the references in classes that it is currently operating on. This includes each class specified on the command line and each class found in a `.zip` file or `.jar` file that is specified on the command line.

The postprocessor cannot detect whether there are `.class` files for which the postprocessor was not called that refer to the renamed package. You must either run the postprocessor on the complete set of class files or modify the Java source of any files that the postprocessor is not annotating.

References to Transient and Persistent Versions of a Class

You might want a class to refer to both the transient and persistence-capable versions of another class.

It is not possible for the postprocessor to determine the references that should be to persistence-capable objects. Because of this, you must code the class so it uses the full path name of the different versions of the class. This is the only way to clarify the version of the class that is wanted. However, this technique works correctly only when you are operating across batches. It does not work when you are within the same batch.

Example

Following is an example of what that means. Suppose you have a utility class called `a.b.C`. You want to have both a transient and a persistence-capable version of `a.b.C`. When you run the postprocessor, you specify `-translatepackage` to create a persistence-capable version called `y.z.C`. Then in another class called `a.b.D`, you try to use both versions of the class. You write source code in `a.b.D` that explicitly refers to `y.z.C` much like the following:

```
int n= y.z.C.countThem()
```

When you try to compile `a.b.D`, compilation can succeed only if you put the annotated classes into the class path of the compiler. Otherwise, the compiler reports an error, because there is no such thing as `y.z.C`. Also, it is not possible for `a.b.C` and `a.b.D` to be in the same batch, because the `-translatepackage` option would apply to `a.b.D`. This would make all of `a.b.D`'s calls go to the persistence-capable version, which is not what you want.

Steps to follow

To use persistence-capable and transient versions of the same class, follow these steps:

1. Create a utility library.

   This is the first batch. This library creates transient versions of the class.

2. Run the postprocessor on the first batch and specify options that put the two different versions of the class in two different packages.

   This step creates the persistence-capable version of the class.

3. Use the library from an application.

   The application is the second batch.

4. Compile the application with the annotated files of the first batch, but not the second batch, in the compiler's class path.

References to Transient Instances of a Persistence-Capable Class

You can use instances of a persistence-capable class in a transient-only manner. No special action is required and the calls to `ObjectStore.fetch()` and `ObjectStore.dirty()` do nothing. There is no need for the unannotated version of the class to be available at run time. To use the annotated version of the class, even if you are using it transiently, the `pro.jar` or `stublib.jar` file must be available in the `CLASSPATH` at run time. If you are using the class only transiently, it can be the `stublib.jar` that is available.

Creating Persistence-Capable Classes with Transient Fields

You can create a persistence-capable class with transient fields. A transient field is a field that is not stored in the database. The postprocessor ignores transient fields. Use the `transient` keyword to create a transient field. For example:

```
class A {
  transient java.awt.Component myVisualizationComponent;
  int myValue;
  ...
}
```

In this class, the `myVisualizationComponent` field is declared to be a transient reference to `java.awt.Component`. The `java.awt` package contains GUI classes that do not lend themselves to being persistence capable.

In your persistence-capable class, you might have transient fields that you want to be able to access outside a transaction. In this situation, you can specify the `-noannotatefield` or `-naf` option for the field when you run the postprocessor. This option prevents access to the specified field from causing `fetch()` and `dirty()` calls on the containing object. Normally, access to a transient field causes `fetch()` or `dirty()` to be called to allow the `postInitializeContents()` and `preFlushContents()` methods to convert between persistent and transient state.

Transient Fields and Serialization

If you have a class that has fields that are declared as transient, this causes the default handling of these fields by object serialization to ignore the fields. If you want them ignored by object serialization and you want them to be stored persistently, specify the `-ignoretransient` option for the class when you run the postprocessor.

On the other hand, there might be a field that must be available for object serialization, but you do not want to store that field in the database. In this situation, specify the `-transientfield` option for the field when you run the postprocessor. This option causes the postprocessor to treat the specified field as though it has a `transient` modifier, even if it does not.

Initialization of Some Transient Fields

In the declaration of a transient field in a persistence-capable class, you might want to initialize the value of the transient field. However, when the postprocessor creates the hollow object constructor for the class, it does not define the constructor to initialize the transient field. This is true even when you specify the `final` keyword. The postprocessor does not initialize such fields because the initialization occurs as inlined code in each of the constructors for the class. For example:

```
private transient final MyField myField = new MyField();
```

The `final` keyword indicates to the postprocessor that initialization is required. However, the initialization code is not readily available and `myField` is not initialized. There are several ways to handle this situation.

You can create the hollow object constructor manually. For example, suppose you define the `MyField` class, which extends the `MyFarm` class, as in the following example:

```
...
public MyField(com.odi.ClassInfo dummyClassInfo) {
  super(dummyClassInfo);
}
```

This requires you also to manually define a hollow object constructor for the `MyFarm` class and for each superclass of the `myFarm` class.

Alternatively, you can remove the `final` qualifier and initialize the transient field in an `IPersistent.postInitializeContents()` method.

If you include an inline initialization of a field declared to be `transient` and `final`, the postprocessor displays an error message and stops processing. If you include an inline initialization of a field declared to be `transient`, but not `final`, the postprocessor warns you about the situation and continues processing. If you determine that you can safely ignore the message, you can turn it off with the `-ignoretransient` option to the postprocessor.

See also Transient Fields in Persistence-Capable Classes on page 130.

Customizing Updated Classes

There are several ways you can customize persistence-capable and persistence-aware annotations: You can implement your own versions of methods that the postprocessor typically adds; you can implement hook methods that PSE Pro calls at specified points; you can define a hollow object constructor in place of the hollow object constructor the postprocessor typically defines; you can also insert your own `fetch()` and `dirty()` calls.

Implementing Customized Methods and Hook Methods

The three methods described next are among the several annotations that the postprocessor adds to persistence-capable classes.

- The `initializeContents()` method loads real values into hollow instances of your persistence-capable class. In other words, hollow objects become active objects with an internal clean state.
- The `flushContents()` method copies values from a modified instance (active persistent object) back to the database. This changes the internal clean or dirty state of the persistent object to the clean state.
- The `clearContents()` method resets the values of an instance to the default values. This changes a clean active object to a hollow object.

Alternatives

If you want to, you can customize the behavior of these methods in the following two ways:

- Implement the method yourself. See Defining Required Methods in the Class Definition on page 209. If you do, the postprocessor does not add the method. However, if you implement any of the three methods listed previously, you must implement all of them. Also, you must define the `ClassInfo` subclass, define an instance of it, and register the instance. This is because the `ClassInfo` instance and the three previous methods must agree on the conventions for field numbering. An example of a program that implements these methods is in the `com/odi/demo/rep` directory in the `Rectangle.java` file. See `com/odi/demo/rep/README.htm`.
- Implement the hook method that corresponds to the method you want to customize. The postprocessor does not annotate hook methods. These hook methods provide a way to perform transient field maintenance. You might also be able to use these methods as an update mechanism for notification about a change:
  - `postInitializeContents()` — If you define this method, PSE Pro calls it immediately after it calls the `initializeContents()` method.
  - `preClearContents()` — If you define this method, PSE Pro calls it just before it calls the `clearContents()` method.
  - `preFlushContents()` — If you define this method, PSE Pro calls it just before it calls the `flushContents()` method.

Warning

The body of a hook method must not call any methods of the class and must not start or end a transaction. This is because the class methods are annotated and, therefore, make calls to `fetch()` and `dirty()`. Such calls in the middle of initializing or writing the object are not allowed because they might cause the virtual machine to encounter a stack overflow.

Sample program with hook methods

Following is an example of a program that implements these hook methods:

```
import com.odi.*;

/**
* PColor provides a persistent representation of colors that can
* be used with the Java AWT package. The java.awt.Color class
* itself cannot be stored persistently, because some of its
* internal state depends on the particular kind of color display
* being used. If a java.awt.Color were created on a computer
* that used a 24-bit-deep color monitor, stored in a database,
* and then retrieved and used on a different computer that had
* a gray-scale monitor, it would not function correctly. PColor
* stores the color value as three integers, and then recreates
* the java.awt.Color object whenever the PColor object is
* brought into Java from persistent storage.
*
* For expository purposes, this example pretends that the value
* of a java.awt.Color object can change after the object is
* created. The real java.awt.Color class is immutable, and so
* the setBlue method below would not work, and the
* preFlushContents method would not actually be needed.
*/

public class PColor {
```

```
/*These instance variables are stored persistently. They
  represent the color value. */
  int red;
  int green;
  int blue;

/*This instance variable is declared transient, so it is not
stored persistently. It is managed by the methods below. */
transient java.awt.Color color;
  PColor(int r, int g, int b) {
    red = r;
    green = g;
    blue = b;
    color = new java.awt.Color(r, g, b);
  }
/*When a PColor is brought into Java from persistent storage,
  the java.awt.Color object is created. Note that this method
  runs after the initializeContents, so that it can use the
  values of the persistent instance variables. */
  public void postInitializeContents() {
    color = new java.awt.Color(red, green, blue);
  }
/*When a PColor is sent out from Java to persistent storage, the
color value from the java.awt.Color object is copied into the
persistent instance variables, so that it will be saved.
Note that this method runs before flushContents, so that it
can set up the values of the persistent instance variables. */

  public void preFlushContents() {
    red = color.getRed();
    green = color.getGreen();
    blue = color.getBlue();
  }
/*When clearContents happens, this method sets the color
instance variable to null, so that this PColor object won't be
stopping the java.awt.Color object from being reclaimed. */

  public void preClearContents() {
    color = null;
  }
/*Equality for PColor objects is the same as equality of the
underlying java.awt.Color objects. */

  public boolean equals(Object obj) {
    if (obj instanceof PColor) {
      return color.getRGB() == ((PColor)obj).color.getRGB();
    }
    return false;
  }
  public java.awt.Color getColor() {
    return color;
  }
  public int getBlue() {
    return color.getBlue();
  }
  public int setBlue(int b) {
    color.setBlue(b);
  }
  /* and so on.... */
}
```

Creating a Hollow Object Constructor

For each persistence-capable class, the postprocessor finds or generates a hollow object constructor. The hollow object constructor takes a single argument whose type is `com.odi.ClassInfo`. Typically, you need not define a hollow object constructor, but you can if you want to.

Why define one?

A reason to define your own hollow object constructor is to initialize transient fields that you want to be usable, even if the `fetch()` method has not been called.

You should avoid performing actions in a hollow object constructor that would cause the object to be fetched. Doing so might cause infinite recursion to occur.

For example, if a class has a persistent `hashCode()` method, it is a bad idea to define a hollow object constructor to register the instances of the class in a hash table. Doing so would cause the `hashCode()` method to be called, which in turn would attempt to fetch the object.

Creation steps

When the postprocessor creates the hollow object constructor, it follows these steps:

1. The postprocessor selects an appropriate superclass hollow object constructor.

   If the superclass has an accessible constructor that takes a single `com.odi.ClassInfo` argument, or if it will have one because the postprocessor adds it during this execution of the tool, the postprocessor uses that constructor. The postprocessor reports an error if it cannot find an accessible constructor.

2. The postprocessor creates a public constructor that
   - Accepts a `com.odi.ClassInfo` argument
   - Invokes the selected superclass constructor
   - Initializes all persistent fields to an appropriate default state that is equivalent to the result of the `clearContents()` method

You can define the hollow object constructor instead of allowing the postprocessor to do it. If you define one, the postprocessor does not generate one.

Optimizing Operations That Retrieve Persistent Objects

Before an application can access the contents of a persistent object, it must call the `ObjectStore.fetch()` method to read the object or the `ObjectStore.dirty()` method to modify the object. These calls make the contents of the object available to your application. The postprocessor inserts these calls in methods of classes that it makes persistence capable or persistence aware. However, the postprocessor might not annotate your code for best performance. You might find that you can improve performance by inserting the `fetch()` and `dirty()` calls yourself.

Caution

If you insert a `fetch()` or `dirty()` call in a method, the postprocessor does not add any additional `fetch()` or `dirty()` calls to that method.

Procedure for Optimizing Operations

Before you add the calls yourself, first allow the postprocessor to add the `fetch()` and `dirty()` calls. Then run and monitor your program. If you want to try to improve performance, add the calls to your source file and recompile. When you run the postprocessor again, it recognizes that the `fetch()` or `dirty()` call is already in place and does not add any `fetch()` or `dirty()` calls to any methods that already contain such a call.

If you do this annotation, you should also add `implements IPersistent` to the definition of any class that is accessed with a `fetch()` or `dirty()` call. When you do this, the compiler can effectively use the multiple overloadings of the `fetch()` and `dirty()` methods, which take `com.odi.IPersistent` arguments. Also, the compiler can generate more efficient code when you declare the class to implement `IPersistent` in your source.

Inlining Code

An important consideration when annotating by hand is that the compiler might inline the code into calling methods. This makes it appear to the postprocessor that the code annotations are in the calling method, which might not be true.

When you are using the JDK `javac` compiler, this occurs when you specify the `-O` (capital O, as in Oslo) option.

To ensure that the postprocessor functions correctly, you must do one of the following:

- Prevent the compiler from inlining code.
- If you add `fetch()` and `dirty()` calls to a method that is a candidate for inlining, also annotate all the methods that call that method. A method is a candidate for inlining if it calls `static`, `final`, or `private` methods, or invokes methods with the `super` qualification construct.

Preventing Fetch of Transient Fields

You might want to avoid the insertion of the `fetch()` call in methods that operate only on transient fields. A strategy for doing this takes advantage of the fact that the postprocessor does not annotate a method if it already includes a `fetch()` or `dirty()` call. If you know that a method operates only on transient fields, you can prevent insertion of the `fetch()` call with code such as the following:

```
try {
  method body goes here
}  catch (SomeRuntimeExceptionThatWillNotOccur) {
    ObjectStore.fetch(this);
}
```

This imposes no execution time and prevents the postprocessor from inserting the `fetch()` method. You can create your own exception, which inherits from `java.lang.RuntimeException`, or select an existing one. The safest approach is to create your own exception so you can be sure that the exception is never signaled.

Performing a Test Run of the Postprocessor

You can run the postprocessor without actually updating any files. The tool performs all processing and error checking and can display messages that indicate what it is doing. This allows you to make corrections before creating the persistence-capable versions of your classes. To perform a test run of the postprocessor, specify the `-nowrite` option on the command line. For example:

```
osjcfp -dest osjcfpout -nowrite classes.jar
```

This command processes all class files in the `.jar` file and displays any error messages. To view information messages from the postprocessor, include the `-verbose` option. For example:

```
osjcfp -dest osjcfpout -nowrite -verbose classes.jar
```

It does not matter where you place the `-nowrite` or `-verbose` option in the command line. Wherever you place them, they apply to all files that the postprocessor processes.

To suppress nonfatal warning messages, specify the `-quiet` option. The `-quiet` and `-verbose` options are mutually exclusive. The last one used on the command line applies to the entire execution. For example, the following line suppresses warning messages during the processing of all specified files because the `-quiet` option follows the `-verbose` option.

```
osjcfp -dest osjcfpout -nowrite -verbose classes.jar -quiet more.jar
```

You can also suppress some, but not all, warnings. Specify the `-quietclass` option followed by the fully qualified name of a class to suppress warnings for that class. Specify the `-quietfield` option followed by the fully qualified name of a field to suppress warnings that pertain to that field. These options apply only to the element whose name immediately follows the option. If the `-verbose` option is also specified, these options take precedence.

Using an Input File

When you are running the postprocessor on a lot of files and specifying many options, the command line can be very long. As a convenience, you can enter the options and file names in a file, then specify the file name as a postprocessor option. Be sure to prefix the file name with the `@` symbol.

Windows

On Windows systems, there is a limit of eight arguments on a command line. Consequently, you usually must use input files on Windows.

Format

You can include comments in the input file. You can place items on different lines and line continuation symbols are not required. Line breaks are treated as white space. Otherwise, enter data in the input file exactly as you would enter it on the command line.

Indicate comments with a # sign. The postprocessor ignores any subsequent characters on the same line as the # sign.

Example

For example, suppose you enter some postprocessor options and files for the postprocessor to operate on in an input file named `optionsAndFiles`. You specify this file as follows:

```
osjcfp @optionsAndFiles
```

You can intersperse input file specifications with options and files that you enter on the command line. For each specified input file, the postprocessor removes any comments from the input file and replaces the input file specification with the data in the input file. The postprocessor then begins to process the command line. For example:

```
osjcfp -dest osjcfpout @file1 -tp old.pack new.pack @file2
```

The postprocessor

1. Replaces `@file1` with the contents of `file1`
2. Replaces `@file2` with the contents of `file2`
3. Executes the command line starting with the `-dest` option

Nesting and wildcards

You cannot nest input file specifications. That is, you cannot include the *`@file_name`* option in an input file. Also, you cannot use wildcards in an input file. The postprocessor does not expand them.

Annotations You Must Add

There are some annotations that the postprocessor either cannot perform or does not perform because of execution performance considerations. You must include these annotations when you code your source files.

Keep in mind that when you add even one `fetch()` or `dirty()` call to a method, the postprocessor recognizes that the method is already annotated and does not add any other `fetch()` or `dirty()` calls to that method. If you do annotate a method, be sure to add all required calls.

This section provides information about the following topics:

- Interfacing with Nonpersistent Methods
- Interfacing with Native Classes
- Annotating Subclasses
- Passing Arrays
- Implementing the Hollow Object Constructor for Some Instance Fields
- Using the Java Reflection API with Persistence-Capable Objects

Interfacing with Nonpersistent Methods

It is possible for a method in a persistence-capable class to pass a persistent object to a nonpersistent method. When this happens, you must ensure that there is a `fetch()` or `dirty()` call for the persistent object before it is passed to the nonpersistent method.

If all access to persistent objects is through annotated methods (methods in persistence-capable or persistence-aware classes), manual annotations are not required. For arrays, there is no way to define a class so arrays of that class can be accessed only by persistence-aware classes. You must be sure to call the `fetch()` or `dirty()` method on a persistent array before passing it to a method in a nonpersistent class.

Interfacing with Native Classes

The postprocessor cannot analyze or annotate native methods. If your code passes a persistent object to a native method, and if the native code might try to access the object other than through annotated methods, be sure to insert a call to `fetch()` or `dirty()` for the persistent object before it is passed. In cases in which native code might access or navigate among persistent objects, you must do one of the following:

- Modify the native code to call `fetch()` or `dirty()` itself.
- Make the necessary `fetch()` and `dirty()` calls before calling the native method.

Annotating Subclasses

After you create a persistence-capable or persistence-aware class, you can define a subclass of that class. Doing so does not make the subclass persistence capable or persistence aware. You must run the postprocessor on the subclass.

If you forget to run the postprocessor on a subclass and if the subclass is reachable from a persistent root, other than through a transient field, PSE Pro might try to migrate instances of the subclass to the database. This attempt causes an error because the subclass is not persistence capable.

Passing Arrays

In your application, you might pass an array to a nonpersistent method when the nonpersistent method is defined as having a parameter of type `java.lang.Object`. In this situation, the postprocessor cannot determine that it should insert `fetch()` or `dirty()` calls for the array in the calling method before passing the array. You must annotate the calling method yourself.

If the called method is declared to accept an array argument, the postprocessor recognizes that a `fetch()` call might be needed and inserts it.

Implementing the Hollow Object Constructor for Some Instance Fields

A class can include nonstatic (instance) fields that contain initializer expressions in their declarations. Postprocessor-generated `ClassInfo` constructors do not run these initializers. Normally, this is not a problem. The constructor allows hollow object initialization and the `initializeContents()` method overwrites these fields when the object is fetched.

However, there might be transient nonstatic fields that have initializer expressions or fields that are treated as transient by your implementation of the `ClassInfo` type and the `initializeContents()` and `flushContents()` methods. In this case, you must manually implement the hollow object constructor or PSE Pro does not run the initializer. It is impossible for the postprocessor to detect such cases, and no warning message can be provided. See Creating a Hollow Object Constructor on page 200.

Using the Java Reflection API with Persistence-Capable Objects

You can use the `java.lang.reflect.Field` class to get and set fields of persistence-capable objects. To do so, you must

- Call the `ObjectStore.fetch()` method for an object before you call any of the `java.lang.reflect.Field` *`get`* methods to get the value of any of the object's fields.
- Call the `ObjectStore.dirty()` method for an object before you call any of the `java.lang.reflect.Field` *`set`* methods to set the value of any of the object's fields.

Class File Postprocessor Limitations

It is possible to cause invalid references when you run the postprocessor and rename the package. In an annotated class, the postprocessor locates and updates class names if they are in field, method, or class references. The postprocessor cannot locate and update string arguments to `Class.forName()` if the name specifies a class whose package has been renamed.

Chapter 9
Generating Persistence-Capable Classes Manually

This chapter provides information about the way to define persistence-capable and persistence-aware classes in your program explicitly without using the automated class file postprocessor supplied with PSE Pro.

Technical Support recommends that you use the automated postprocessor. See Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171. For information on which Java-supplied classes are persistence capable, see Java-Supplied Persistence-Capable Classes on page 261.

You might choose the manual method if you want to

- Manually optimize the code
- Perform translation between nonpersistent objects and a custom persistent representation

You can partially manually annotate a class, then run the postprocessor to insert the remaining required annotations.

You must explicitly postprocess or manually annotate each class that you want to be persistence capable. The capacity for an object to be stored in a databases is not inherited when you subclass a persistence-capable class.

Contents

This chapter discusses the following topics:

Explicitly Defining Persistence-Capable Classes

Follow these steps to annotate your program so that classes you define are persistence capable:

1. Define your class to implement the `IPersistent` interface. See page 208.
2. In the class definition, define the required fields. See page 208.
3. In the class definition, define the required methods. See page 209.
4. In the class definition, define accessor methods so that they make the appropriate ObjectStore.fetch() and ObjectStore.dirty() method calls. See page 211.
5. If required, define a class that extends the ClassInfo class. See page 212.

   Interfaces never require `ClassInfo` classes.

   If you will be running your application in an environment that allows the unrestricted use of the Java reflection API, public or abstract classes with hollow object constructors do not require `ClassInfo` classes.
6. For any `ClassInfo` subclasses you define, create an instance of the `ClassInfo` subclass. Only one instance of this subclass is ever needed.
7. Call the static `get()` method on `ClassInfo`. (Typically, this is in static initializer code for the manually annotated class.) See page 212.

Some Java-supplied classes are persistence capable. Others are not persistence capable and cannot be made persistence capable. A third category of classes can be made persistence capable, but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 261.

About interfaces

Interfaces are always persistence capable. You must specify them when you run the postprocesor, but other than that, you need not do anything to make an interface persistence capable.

Implementing the IPersistent Interface

Every persistence-capable class must implement the `IPersistent` interface or be a subclass of a class that implements it. As with any interface, every method defined in the `IPersistent` interface must be defined in a class that implements `IPersistent`. If you want to rely on the postprocessor to insert the missing methods for you, you must not explicitly implement the `IPersistent` interface.

Defining the Required Fields

The following code must be in your class definition. You can add this code yourself, or you can run the postprocessor to add it.

```
transient private com.odi.imp.ObjectReference ODIRef;
transient public byte ODIObjectState;
```

The `ODIRef` field stores a reference. The `ODIObjectState` field holds some object state bits. The underlying run-time classes in PSE Pro access these fields through the `IPersistent` accessor methods, as needed.

Defining Required Methods in the Class Definition

This section describes the methods that must be defined in a class that implements the `IPersistent` interface.

Define the `initializeContents()` method to load real values into hollow instances of your class. This changes a hollow object to an active object. PSE Pro provides methods on the `GenericObject` class that retrieve each `Field` type. Be sure to call the correct methods for the fields in your persistent object. There is a separate method for obtaining each type of `Field` object. PSE Pro calls the `initializeContents()` method as needed. The method signature is

```
public void initializeContents(GenericObject genObj)
```

Following is an example:

```
public void initializeContents(GenericObject handle) {
  name = handle.getStringField(1, PCI);
  age = handle.getIntField(2, PCI);
  children = (Person[])handle.getArrayField(3, PCI);
}
```

If the class you are annotating implements `IPersistent` through a superclass, you must also initialize superclass fields by invoking `initializeContents()` on the superclass.

Define the `flushContents()` method to copy values from a modified instance (active persistent object) back to the database. This method changes an active clean or dirty object to an active clean object. PSE Pro provides methods on the GenericObject class that set each `Field` type. Be sure to call the correct methods for the fields in your persistent object. There is a separate method for setting each type of `Field` object. PSE Pro calls the `flushContents()` method as needed. The method signature is

```
public void flushContents(GenericObject genObj)
```

Following is an example:

```
public void flushContents(GenericObject handle) {
  handle.setClassField(1, name, PCI);
  handle.setIntField(2, age, PCI);
  handle.setArrayField(3, children, PCI);
}
```

If the class you are annotating implements `IPersistent` through a superclass, you must also flush superclass fields by invoking `flushContents()` on the superclass.

Define the `clearContents()` method to reset the values of an instance to the default values. This method changes an active clean object to a hollow object. This method must set all reference fields that referred to persistent objects to null. PSE Pro calls this method as needed. The method signature is

```
public void clearContents()
```

Following is an example:

```
public void clearContents() {
  name = null;
```

```
  age = 0;
  children = null;
}
```

If the class you are annotating implements `IPersistent` through a superclass, you must also clear superclass fields by invoking `clearContents()` on the superclass.

Field accessor methods

The following accessor methods must be in the class definition:

- `public ObjectReference ODIgetRef()`
- `public void ODIsetRef(ObjectReference` *`objRef`*`)`
- `public byte ODIgetState()`
- `public void ODIsetState(byte` *`state`*`)`

If you do not want to define them, you can run the postprocessor to insert them for you, but you must not declare the class to implement `IPersistent.` However, if you explicitly define an `ODIget`*`xxx`*`()` method, you must explicitly define its associated `ODIset`*`xxx`*`()` method. Likewise, if you explicitly define an `ODIset`*`xxx`*`()` method, you must explicitly define its associated `ODIget`*`xxx`*`()` method.

If you add the code yourself, it must look like the following:

```
public com.odi.imp.ObjectReference ODIgetRef() {
  return ODIRef;
}

public void ODIsetRef(com.odi.imp.ObjectReference objRef) {
  ODIRef = objRef;
}

public byte ODIgetState() {
  return ODIObjectState;
}

public void ODIsetState(byte state) {
  ODIObjectState = state;
}
```

Implementing the IPersistentHooks Interface

There are times when you might want a persistence-capable class to maintain transient information in parallel with persistent information. The `IPersistentHooks` interface allows you to do this.

If a persistence-capable class implements the `IPersistentHooks` interface, PSE Pro calls the `IPersistentHooks` methods that are defined when it calls the corresponding methods defined in the `IPersistent` interface.

As with any interface, every method defined in the `IPersistentHooks` interface must also be defined in the class that explicitly implements the `IPersistentHooks` interface.

If you explicitly declare that a class implements the `IPersistentHooks` interface without providing all the definitions declared in the interface, you receive a compilation error. However, if you define some methods, but not all of them, and

you do not explicitly declare that the class implements the `IPersistentHooks` interface, you can use the postprocessor to insert the missing methods and the interface declaration.

Hook methods

The following methods must also be in the class definition. You can define them as methods with empty bodies. If you do not define them and your class does not explicitly implement the `IPersistentHooks` interface, you can use the postprocessor to add these methods with empty bodies.

- `postInitializeContents()` is called by PSE Pro immediately after it calls the `initializeContents()` method.
- `preFlushContents()` is called by PSE Pro immediately before it calls the `flushContents()` method.
- `preClearContents()` is called by PSE Pro immediately before it calls the `clearContents()` method.
- `preDestroyPersistent()` is called by PSE Pro after your application calls the `ObjectStore.destroy()` method and before PSE Pro destroys anything.
- `postPersisted()` is called by PSE Pro after it assigns a database location to a new persistent object and after it calls the `flushContents()` method.

Making Object Contents Accessible

In each class that you want to be persistence capable, you must annotate your class definition to include calls to the `ObjectStore.fetch()` and `ObjectStore.dirty()` methods. It does not matter whether the class explicitly implements `IPersistent` or inherits from a class that implements `IPersistent`. These calls are required for the class to be persistence capable.

With some exceptions, before your application can access the contents of an object, it must call the

- `ObjectStore.fetch()` method on the object to read its contents
- `ObjectStore.dirty()` method on the object to modify its contents

Calls to fetch() or dirty()

Your application calls the method and passes an object whose contents you want to access. This makes the contents of the object available. Modify the methods that reference nonstatic fields to call the `ObjectStore.fetch()` and `ObjectStore.dirty()` methods as needed. While this step is not mandatory, it does provide a systematic way of ensuring that the application calls the `fetch()` or `dirty()` method before accessing or updating object contents.

Remember that you can add some annotations and run the postprocessor to add other annotations. You might want to define the required methods and the `ClassInfo` subclass, but let the postprocessor insert the required `fetch()` and `dirty()` calls.

Exceptions

You need not call the `fetch()` or `dirty()` method on instances of primitive wrapper classes (see Description of Java-Supplied Persistence-Capable Classes on page 261). If you do call `fetch()` or `dirty()` on these objects, nothing happens and processing continues.

You need not call the `fetch()` or `dirty()` method on instances of `java.lang.String`. If you call `fetch()` on instances of `java.lang.String`, nothing happens. If you call `dirty()` on instances of `java.lang.String`, PSE Pro signals `ObjectException`.

Defining a ClassInfo Subclass

If required, define a public class that inherits from the `ClassInfo` class. (See page 215 for requirements.) You must define this class in a separate file. If you plan to use the postprocessor to insert any annotations, the name of this class must be one of the following:

- The name of the persistence-capable class, followed by `ClassInfo`, for example, `PersonClassInfo`
- The suffix specified with the `-classinfosuffix` option to the postprocessor

In each `ClassInfo` subclass definition, you must include the following methods.

Define a `create()` method to create instances of your persistence-capable class with default field values:

```
public IPersistent create() {return new Person(this);}
```

This should call a constructor, referred to as a hollow object constructor, that leaves fields in the default state. For an abstract class, the `create()` method can return null.

Define the public `getClassDescriptor()` method to obtain the class object for your class. For example:

```
public Class getClassDescriptor()
  throws ClassNotFoundException {
  return Class.forName("com.odi.demo.people.Person"); }
```

Define the public `getFields()` method to allow access to the names and types of the fields of the class. For example:

```
public Field[] getFields() { return fields; }
private static Field[] fields = {
  Field.createString("name"),
  Field.createInt("age"),
  Field.createClassArray("children", "Person", 1)
};
```

The definition of the `getFields()` method can specify create methods for fields that are not in the class definition and can omit create methods for fields that are in the class definition.

Example of a Manually Annotated Persistence-Capable Class

Following is an example of a definition of a manually annotated persistence-capable class. Three consecutive periods indicate lines from a complete program that have been omitted here because they are not pertinent to creating a persistence-capable class.

Class definition

```
package com.odi.demo.people;
import com.odi.*;
```

```
// Define a class that implements IPersistent:

class Person implements IPersistent {

  // Fields:

  String name;
  int age;
  Person children[];
  // Other fields ...

  // Constructor:

  public Person(String name, int age, Person children[]) {
    this.name = name; this.age = age; this.children = children;
  }

  // Hollow object constructor:

  public Person(ClassInfo info) { }

  // Accessor methods that have been modified to call
  // the fetch() and dirty() methods:

  public String getName() {ObjectStore.fetch(this);
      return name; }
  public void setName(String name) {ObjectStore.dirty(this);
       this.name = name; }
  public int getAge() {ObjectStore.fetch(this); return age; }
  public void setAge(int age) {ObjectStore.dirty(this);
       this.age = age; }
  public Person[] getChildren() {ObjectStore.fetch(this);
      return children; }
  public void setChildren(Person children[]) {
      ObjectStore.dirty(this); this.children = children;
  }
  // Other methods ...

  // Additions required for PSE Pro:

  // Define the initializeContents() method to load real
  // values into hollow persistent objects, which makes
  // them active persistent objects:

  public void initializeContents(GenericObject handle) {
    name = handle.getStringField(1, myClassInfo);
    age = handle.getIntField(2, myClassInfo);
    children = (Person[])handle.getArrayField(3, myClassInfo);
  }

  // Define the flushContents() method to copy the
  // contents of a persistent object to the database:

  public void flushContents(GenericObject handle) {
    handle.setClassField(1, name, myClassInfo);
    handle.setIntField(2, age, myClassInfo);
    handle.setArrayField(3, children, myClassInfo);
  }
```

```
  // Define the clearContents() method to reset the values
  // of a persistent instance to the default values.
  // This method must set all reference fields that
  // referred to persistent objects to null:

  public void clearContents() {
    name = null;
    age = 0;
    children = null;
  }

  // Define the ODIRef and ODIObjectState fields and
  // their accessor methods.
  transient private com.odi.imp.ObjectReference ODIRef;
  transient public byte ODIObjectState;
  public com.odi.imp.ObjectReference ODIgetRef() {
      return ODIRef;
  }
  public void ODIsetRef(com.odi.imp.ObjectReference objRef) {
      ODIRef = objRef;
  }
  public byte ODIgetState() {
      return ODIObjectState;
  }
  public void ODIsetState(byte state) {
      ODIObjectState = state;
  }

  // Create an instance of the subclass of ClassInfo and
  // register that instance:

  static ClassInfo myClassInfo =
      ClassInfo.get("com.odi.people.Person");
}
```

ClassInfo definition

In a separate file, define the subclass of the `ClassInfo` class if its definition is required. For example:

```
// Define the subclass of ClassInfo. A recommended naming
// convention is to prefix the name of your persistence-capable
// class to "ClassInfo".

package com.odi.demo.people;

import com.odi.*;

public class PersonClassInfo extends ClassInfo {

  // Define a create() method to create instances of your
  // class with default field values. The method
  // calls the hollow object constructor and passes this,
  // which is an instance of the ClassInfo subclass:

  public IPersistent create() { return new Person(this); }

  // Define these public methods to provide access to
  // the name of the persistence-capable class, the name of its
  // superclass, and the names of its fields.
  // The array returned by getFields() must contain the
  // fields in the order of their field numbers.
```

```
  public Class getClassDescriptor()
    throws ClassNotFoundException {
  return Class.forName("com.odi.demo.people.Person"); }
    public Field[] getFields() { return fields; }
    private static Field[] fields = {
    Field.createString("name"),
    Field.createInt("age"),
    Field.createClassArray(
    "children", "com.odi.demo.People.Person", 1)
  };

}
```

It does not matter whether the `ClassInfo` class explicitly implements `IPersistent` or inherits from a class that implements `IPersistent`.

`ClassInfo` is an abstract class for managing schema information for persistence-capable classes. PSE Pro requires the schema information to manage the object. If you do not explicitly define a `ClassInfo` class, PSE Pro uses the Java reflection API to create the needed information at run time.

After you perform the steps described in this section, you can store instances of your class in a database.

PSE Pro does not let you store `final` instance variables persistently. This is because it is not possible to write the `initializeContents()` and `clearContents()` methods to handle `final` instance variables correctly.

Additional Information About Manual Annotation

This section provides additional information about manually annotating a class to be persistence capable. It discusses the following topics:

- Defining a hashCode() Method
- Defining a clone() Method
- Working with Transient-Only and Persistent-Only Fields
- Defining Persistence-Aware Classes
- Following Postprocessor Conventions
- Annotating Abstract Classes

Defining a hashCode() Method

Every class inherits from the `Object` class, which defines the `hashCode()` method and provides a default implementation. For a persistent object, this default implementation often returns a different value for the same persistent object (the object on the disk) at different times. This is because PSE Pro fetches the persistent object into different Java objects at different times (in different transactions or different invocations of Java).

This is not a problem if you never use the object as a key in a persistent hash table or other structure that uses the `hashCode()` method to locate objects. If you do use the object as a key, the hash table or other structure that relies on the `hashCode()` method might become corrupted when you bring the objects back from the database.

To resolve this problem, you can define your own `hashCode()` method and base it on the contents of the objects so it returns the same thing every time. The signature of this method must be

```
public int hashCode()
```

Defining a clone() Method

If your persistence-capable class implements the `Cloneable` interface, your class must define a `clone()` method. This `clone()` method must ensure that it correctly initializes and checks the `ODIRef` and `ODIObjectState` fields when it performs a clone operation. For new cloned objects, your application should initialize `ODIRef` to null and `ODIObjectState` to zero.

Working with Transient-Only and Persistent-Only Fields

The definition of the `ClassInfo.getFields()` method returns an array of `com.odi.Field` instances. There is one element for each field that you want to store and retrieve in a persistent object. PSE Pro does not require an exact match between each field in the Java class definition and each field array element returned by the `getFields()` method. Furthermore, fields listed in the `getFields()` return value need not directly represent fields in the class. They can represent state from which values for fields in the class are synthesized.

Transient-only fields

A persistence-capable Java class can define a field that does not appear in the list of fields returned by the `ClassInfo.getFields()` method. Such a field is a transient-only field. The `initializeContents()` method that is associated with the class can be used to initialize transient-only fields based on persistent state. For example:

```
class A {
  transient java.awt.Component myVisualizationComponent;
  int myValue;
    ...
  }
```

In this class, the `myVisualizationComponent` field is declared to be a transient reference to `java.awt.Component`. The `java.awt` package contains GUI classes that do not lend themselves to being persistence capable.

Number of fields

The number of nonstatic, nontransient declared fields in the class should generally be equal to the number of fields reported by the `getFields()` method, unless the `flushContents()` and `initializeContents()` methods are written to combine or split fields. If they are so written, you can define an arbitrary mapping of persistent fields to Java instance fields. For example:

```
class Some {
  int a;
  int b;
  int aPlusb;
```

```
  initializeContents(GenericObject, go) {
    a=go.getField(1, SomeClassInfo);
    b=go.getField(2, SomeClassInfo);
    c=a+b;
  }
  ...
}
```

In a separate file:

```
public class SomeClassInfo
  static Field[] fields=
{  field.createInt"a");
  field.createInt("b");
}
```

Persistent-only fields

The list of `Field` objects returned by the `getFields()` method might include one or more fields that are not in the Java class definition. Such fields are persistent-only fields. The `flushContents()` method associated with the class must set the field value in the generic object based on other fields of the class.

Variable initializers

If you manually annotate a class, you should avoid using variable initializers to initialize persistent fields of persistence-capable objects. Instead, perform the initialization in the constructor. This is because the values computed by the variable initializer expression typically are overwritten by the `com.odi.IPersistent.initializeContents()` method. When an object is actually fetched from the database, the fields are initialized with their correct persistent values.

Example

An example of how you might use transient-only and persistent-only fields is in the `demo` directory that is included in PSE Pro. In the `rep` example, `Rectangle.a` and `Rectangle.b` are transient-only fields, while `ax`, `ay`, `bx`, and `by` are persistent-only fields. Following is the part of the example that shows this:

```
package com.odi.demo.rep;

/**
* A Rectangle has two Points, representing its upper-left
* and lower-right corners. However, its persistent
* representation is formed by storing the x and y coordinates of
* the two points, rather than the points themselves. This
* demonstrates the control that the definer of a persistent
* class has over the persistent representation. Note that
* Identity of the Point objects is not preserved, since thePoint *
objects are not persistent objects. */


import com.odi.*;

public class Rectangle implements IPersistent {

  transient private com.odi.imp.ObjectReference ODIref;
  transient public byte ODIobjectState;

  transient Point a;
  transient Point b;
```

```
  static ClassInfo classInfo
    = ClassInfo.register(new RectangleClassInfo());

  public com.odi.imp.ObjectReference ODIgetRef() {
    return ODIref;
  }

  public void ODIsetRef(com.odi.imp.ObjectReference objRef) {
    ODIref = objRef;
  }

  public byte ODIgetState() {
    return ODIobjectState;
  }
  public void ODIsetState(byte state) {
    ODIobjectState = state;
  }

  Rectangle(Point a, Point b) {
    this.a = a;
    this.b = b;
  }

  void describe() {
    System.out.println("Rectangle with two points:");
    a.describe();
    b.describe();
  }
  /* Annotations for persistence. */

  Rectangle(ClassInfo ignored) {}

  public void initializeContents(GenericObject handle) {
    a = new Point(handle.getIntField(1, classInfo),
      handle.getIntField(2, classInfo));
    b = new Point(handle.getIntField(3, classInfo),
      handle.getIntField(4, classInfo));
  }

  public void flushContents(GenericObject handle) {
    handle.setIntField(1, a.x, classInfo);
    handle.setIntField(2, a.y, classInfo);
    handle.setIntField(3, b.x, classInfo);
    handle.setIntField(4, b.y, classInfo);
  }

  public void clearContents() {
    a = null;
    b = null;
  }
  /* This class is never used as a persistent hash key. */
  public int hashCode() {
    return super.hashCode();
  }
}
```

In a separate file:

```
public class RectangleClassInfo extends ClassInfo
{
```

```
public IPersistent create() { return new Rectangle(this); }
public Class getClassDescriptor() throws
  ClassNotFoundException {
  return Class.forName("com.odi.demo.rep.Rectangle");
}

public Field[] getFields() { return fields; }
private static Field[] fields =
{ Field.createInt("ax"),
  Field.createInt("ay"),
  Field.createInt("bx"),
  Field.createInt("by"), };}
```

Defining Persistence-Aware Classes

A persistence-aware class is a class whose instances

- Can operate on persistent objects
- Cannot be stored in a database

For a class to be persistence aware, you must annotate it so that it includes calls to the `ObjectStore.fetch()` and `ObjectStore.dirty()` methods. The `fetch()` method makes the contents of a persistent object available to be read. The `dirty()` method makes the contents of a persistent object available to be modified.

To make a class persistence aware, modify each method that references

- Nonstatic fields of persistence-capable classes
- Array elements of arrays that might be persistent

Modify each method so that it calls the `ObjectStore.fetch()` or `ObjectStoret.dirty()` method. This call must be before any attempt to access the contents of the persistent object. The `fetch()` and `dirty()` methods make the contents of persistent objects available.

A persistence-aware class includes the `fetch()` and `dirty()` annotations. It does not include the other annotations that are required for a class to be persistence capable.

Following Postprocessor Conventions

If you plan to define all required annotations explicitly, you need not be concerned with postprocessor conventions. However, if you plan to insert some annotations explicitly and use the postprocessor to insert other annotations, you must follow these postprocessor conventions.

- The name of the `ClassInfo` subclass must have the following format:

  `class_name``ClassInfo`

  For example, if you define the `Boat` class, the name of the associated subclass of `ClassInfo` must be `BoatClassInfo`.

- In the `ClassInfo` subclass definition, when you define the hollow object constructor, it must take a single argument of type `ClassInfo`. See page 214.

Annotating Abstract Classes

Persistence-capable classes and their superclasses, even if they are abstract, must each have a corresponding `ClassInfo` subclass. But an application does not create instances of abstract classes, so you cannot write the required `create()` method in the `ClassInfo` subclass in the usual way. Define the `create()` method so that it returns null. Because this method is never called, it is safe to define it this way.

Now suppose you define the following two classes:

```
abstract class Y {
  int yValue;
  abstract void doSomething();
}

class X extends Y {
  float xValue;
  void doSomething() {}
}
```

Class `Y` must have an associated `ClassInfo` subclass and class `X` must have an associated `ClassInfo` subclass. The `ClassInfo` subclass associated with `X` does not extend the `ClassInfo` subclass associated with `Y`.

In the `ClassInfo` subclass for `X`, the `Field` array must include only those fields defined explicitly in `X`; `XClassInfo.getFields()` must report only the immediate persistent fields in `X`. The `ClassInfo` subclass for `Y` defines a `Field` array that contains the fields explicitly defined in `Y`.

Creating and Accessing Fields in Annotations

As part of the process of manually defining a class that is persistence capable, the required annotations must, among other things,

- Define an `initializeContents()` method in the persistence-capable class
- Define a `flushContents()` method in the persistence-capable class
- Define a `getFields()` method in the `ClassInfo` subclass

To define these methods correctly, you must know how PSE Pro makes persistent objects accessible and the methods that are available to create and access individual fields in an object. To help you do this, this section discusses the following topics:

- Making Persistent Objects Accessible
- Creating Fields
- Getting and Setting Generic Object Field Values
- Methods for Creating Fields and Accessing Them in Generic Objects

Making Persistent Objects Accessible

The `ObjectStore.fetch()` method makes the contents of a persistent object available to be read by an application. The `ObjectStore.dirty()` method makes the contents of a persistent object available to be updated by an application.

To execute a `fetch()` or `dirty()` call, PSE Pro first checks whether a `fetch()` or `dirty()` call was already invoked on the object in the current transaction. If it was, PSE Pro does nothing and the program continues. If it was not, PSE Pro executes the `fetch()` or `dirty()` call, as required.

When PSE Pro retrieves a persistent object, it calls the `initializeContents()` method that you defined. The `initializeContents()` method calls methods on `GenericObject` to obtain the field values for the persistent object. The result is that your program has access to the desired data.

PSE Pro provides the `GenericObject` class for transferring data between a database and a Java application or applet. A generic object represents an object's data as it is stored in the database. A generic object is a temporary buffer that PSE Pro uses while it is copying data from the database into a persistent object or writing data into the database from a persistent object. PSE Pro creates instances of `GenericObject` as needed. You do not define subclasses of `GenericObject`, nor do you create instances of `GenericObject`.

For an object that was not already retrieved, PSE Pro copies the contents of the object from the database into the `GenericObject` instance. It then passes this instance to the `initializeContents()` method defined in the persistence-capable class.

Suppose you called the `dirty()` method on a persistent object and modified it. To update the object in the database, commit the transaction. This causes PSE Pro to create an instance of `GenericObject` to hold the contents of your object. Then PSE Pro calls the `flushContents()` method that you defined when you defined the persistence-capable class.

The `flushContents()` method must call methods on the `GenericObject` instance that store the object's field values in the generic object. PSE Pro calls the `flushContents()` method as needed to copy the new contents of the object into the database.

Creating Fields

PSE Pro provides the `Field` class to represent a Java field in a persistent object. When you define a persistence-capable class, you must define a `getFields()` method in the required `ClassInfo` subclass. This method provides a list of the nonstatic fields (also called instance variables) whose values are being stored and retrieved.

Description of getFields()

The `getFields()` method must return an array that contains the nonstatic persistent object fields. The order in which they appear in the array implies their associated field numbers. This array must include only those fields defined in the persistence-capable class and not any inherited fields.

Field numbers represent the position of a nonstatic field within the list of all nonstatic fields defined for the class and its superclasses. The first field has field number 1. (Note that the first field number is not 0.)

Order of fields

When you define the `getFields()` method in the `ClassInfo` subclass, you determine the order and, therefore, the number of each field even though you do not explicitly assign any numbers. PSE Pro assigns the numbers according to the order in which the values are returned from the field create methods defined in the `getFields()` method. The field numbers are consecutive with no gaps. For example:

Example

```
public Field[] getFields() {return fields;}

  private static Field[] fields = {
    Field.createString("name"),
    Field.createInt("age"),
    Field.createClassArray("children",
      "com.odi.demo.people.Person", 1)
  };
```

The previous definition causes PSE Pro to associate `1` with the `name` field, `2` with the `age` field, and `3` with the `children` field.

When you define the `initializeContents()` and `flushContents()` methods, you must specify the correct field number for each field that the methods get and set.

Creation methods

The `Field` class provides a create method for each Java data type. Minimally, the create methods on the `Field` object

- Return the created `Field` object
- Take a `String` parameter that specifies the name of the field

There are separate create methods for singleton and array fields of each primitive type. There are also string fields, class fields, and interface fields. The complete list of field create methods is in Methods for Creating Fields and Accessing Them in Generic Objects on page 223.

Getting and Setting Generic Object Field Values

As described earlier, PSE Pro provides the `GenericObject` class to transfer objects between the database and an application. Consequently, when you define a persistence-capable class, you must define the `initializeContents()` method to retrieve values from fields in instances of `GenericObject` and the `flushContents()` method to set values in fields of instances of `GenericObject`.

When you define the `initializeContents()` and `flushContents()` methods, you must use a method that is appropriate for the type of each field in the instance of `GenericObject`. For example, for each character field, you must use the

- `getCharField()` method in the `initializeContents()` method
- `setCharField()` method in the `flushContents()` method

There is a different method for getting and setting each Java type. To get or set an array of any type, you define the `getArrayField()` and `setArrayField()` methods, respectively. In the `initializeContents()` method, be sure to call the methods that

get the values. In the `flushContents()` method, be sure to call the methods that set the values. The methods that get and set fields in a generic object are listed in the table in Methods for Creating Fields and Accessing Them in Generic Objects on page 223.

Methods for Creating Fields and Accessing Them in Generic Objects

Kind of Java Field	*Code*	*Method That Operates on It*

Single byte	`byte`	`Field.createByte()`
		`GenericObject.getByteField()`
		`GenericObject.setByteField()`

Array of bytes	`byte[]`	`Field.createByteArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single character	`char`	`Field.createChar()`
		`GenericObject.getCharField()`
		`GenericObject.setCharField()`

Array of characters	`char[]`	`Field.createCharArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single 16-bit integer	`short`	`Field.createShort()`
		`GenericObject.getShortField()`
		`GenericObject.setShortField()`

Array of 16-bit integers	`short[]`	`Field.createShortArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single 32-bit integer	`int`	`Field.createInt()`
		`GenericObject.getIntField()`
		`GenericObject.setIntField()`

Array of 32-bit integers	`int[]`	`Field.createIntArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single 64-bit integer	`long`	`Field.createLong()`
		`GenericObject.getLongField()`
		`GenericObject.setLongField()`

Array of 64-bit integers	`long[]`	`Field.createLongArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single 32-bit floating-point number	`float`	`Field.createFloat()`
		`GenericObject.getFloatField()`
		`GenericObject.setFloatField()`

Array of 32-bit floating-point numbers	`float[]`	`Field.createFloatArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single 64-bit floating-point number	`double`	`Field.createDouble()`
		`GenericObject.getDoubleField()`
		`GenericObject.setDoubleField()`

Kind of Java Field	*Code*	*Method That Operates on It*

Array of 64-bit floating-point numbers	`double[]`	`Field.createDoubleArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single Boolean value	`boolean`	`Field.createBoolean()`
		`GenericObject.getBooleanField()`
		`GenericObject.setBooleanField()`

Array of Boolean values	`boolean[]`	`Field.createBooleanArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Single string value	`String`	`Field.createString()`
		`GenericObject.getStringField()`
		`GenericObject.setStringField()`

Array of string values	`String[]`	`Field.createStringArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Enum		`Field.createEnum()`
		`GenericObject.getEnumField()`
		`GenericObject.setEnumField()`

A class		`Field.createClass()`
		`GenericObject.getClassField()`
		`GenericObject.setClassField()`

A class array		`Field.createClassArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

An interface		`Field.createInterface()`
		`GenericObject.getInterfaceField()`
		`GenericObject.setInterfaceField()`

An interface array		`Field.createInterfaceArray()`
		`GenericObject.getArrayField()`
		`GenericObject.setArrayField()`

Chapter 10
Using the Java Dynamic Data (JDD) Classes

The Java Dynamic Data (JDD) classes provide the API for creating, storing, and accessing persistent data, based on type information (schema) that the application defines at run time. JDD is designed for applications that model dynamic data and require greater flexibility when defining and redefining types than is available with persistence-capable Java classes.

This chapter explains how JDD works and shows how to use the JDD classes to perform common database operations.

Contents

This chapter covers the following topics:

Note

To compile and run a Java program that uses the JDD classes, you must set the `CLASSPATH` environment variable to include the location of the JDD `.jar` file.

An Overview of JDD

Three concepts are central to understanding JDD: *types*, *attributes*, and *entities*. These concepts are analogous to Java's *classes*, *fields*, and *objects*. Just as a Java class defines a set of fields that can be assigned values in an object of the class, so a JDD type defines a set of attributes that can be assigned values in an entity of the type.

The difference between Java and JDD is that, in Java, the definition of classes and their fields must occur *before* run time — during program development. In JDD, on the other hand, the definition of types and their attributes occurs dynamically at run time. Furthermore, JDD enables you to add, change, or remove types and their attributes at run time — without having to run the postprocessor, perform schema evolution, restart the Java virtual machine, or even begin a new transaction. In fact, JDD makes it possible to create a new type, add attributes to the type, create entities of the type, assign attribute values in the entities, and then perform queries — all in

the same transaction. See A Simple JDD Application on page 230 for an example program that uses JDD to perform these tasks within a single transaction.

The following sections provide more information about JDD types, attributes, and entities.

Types

A type is an object of the class `Type` that defines the attributes that can be accessed for entities of the type. Once you have created a type by invoking the `Type()` constructor, you can add attributes and create entities of the type.

Types have multiple inheritance; each type can have multiple subtypes and multiple supertypes.

Each type also maintains an extent of its entities and all of its subtypes' entities. This feature allows you to access entities through their type, either directly or by querying the type. A query on a type executes through the type's extent, which includes the extent of each of its subtypes.

Attributes

An attribute is an object of the class `Attribute`. You create attribute objects when you define a type's attributes, and you assign attribute values in the entities of the type. The different subclasses of `Attribute` — for example, `IntAttribute` and `EntityAttribute` — determine the types of values that can be assigned to attributes. These subclasses provide typed versions of accessor methods — for example, `get()` and `put()` — for type safety.

Methods in the `Attribute` class and its subclasses enable you to do the following:

- Set default values.
- Add constraints, including (for numeric attributes) maximum and minimum values.
- Declare an attribute as transient.
- Make an attribute immutable. Once the value of an immutable attribute is assigned, it cannot be changed.
- Make an attribute required. A required attribute must be assigned a value for each entity of the type.

Relationships

You can also use attributes to represent bidirectional relationships. The `Relationship` interface and the classes that implement it enable you to model three types of relationships:

- One-to-one
- One-to-many
- Many-to-many

To create a relationship, you invoke the constructor for the appropriate relationship class. The constructor creates attributes in the types on both sides of the relationship. Each attribute references the other side of the relationship.

For example, the `OneToMany()` constructor could be used to create a one-to-many relationship between two types, `Department` and `Employee`. The constructor would create one attribute in the `Department` type and the other in the `Employee` type. The attribute for `Department` could contain references to all `Employee` entities who are members of a department, and the attribute for `Employee` could reference the `Department` entity.

JDD maintains referential integrity on both sides of the relationship. To continue with the preceding example, if you were to add or remove an `Employee` entity, JDD would automatically update the attribute in the appropriate `Department` entity.

For more information, see Relationships on page 232.

Entities

An entity is an object of the `Entity` class. You create an entity as belonging to a type, and assign values to an entity's attributes, which are defined by the type. Later, when you add new attributes to the type, you can assign them values in the existing entities. You can also change an entity's type.

To make an entity available to a query, you must add it to the extent of the entity's type — by calling the `addToExtent()` method on the type. You can get a list of all the entities in the extent of a type — exclusive of its subtypes — by invoking the `entities()` method on the type. You can also get a list of all entities in the extent of the type and its subtypes, by invoking the `extent()` method.

Basic JDD Tasks

This section describes how to use the JDD API to perform basic database operations with JDD, including

- Defining Types and Their Attributes
- Creating Entities of a Type
- Querying a Type

Most of the code examples shown in these sections are from the sample program listed at the end of this section; see A Simple JDD Application on page 230.

Defining Types and Their Attributes

Before storing any entities in a database, you must first create their types. To create a type, invoke the `Type()` constructor, as in the following example:

```
Type Car = new Type(db, "Car");
```

The `db` argument is the database that you previously opened or created. The `Car` argument is the name of the new type, which is also the object returned by the constructor. To make `Car` a subtype of an existing type (for example, `Vehicle`), you specify an object of the supertype as the first argument, as in the following example:

```
Type Car = new Type(Vehicle, db, "Car");
```

After creating a type, you can create its attributes. It is not necessary that you create all attributes that the type will ever require. In fact, the value of JDD is that it lets you add attributes on an ongoing basis, as dictated by the changing data model. Likewise, you can remove attributes whenever the need arises.

To create an attribute, you invoke the constructor for one of the subclasses of the `Attribute` class. The choice of the subclass depends on the type of values you will be storing in the attribute; for example, to store floating-point values, invoke the `DoubleAttribute()` constructor.

The following code defines three attributes in the `Car` type — `make`, `color`, and `year`:

```
StringAttribute make = new StringAttribute(Car, "make");
StringAttribute color = new StringAttribute(Car, "color");
IntAttribute year = new IntAttribute(Car, "year");
```

Indexing

If you will be performing query operations on the type, you can optimize queries by adding indexes. As explained in Types on page 226, you perform queries on a type, which maintains an extent of entities. To add an index, call the `addIndex()` method on the type, specifying the name of the attribute to be used as the key. The attribute name (a string) must be preceded by the dollar-sign character (`$`). You can cascade through a hierarchy of attributes by preceding each name with the `$` character; for example, `$a.$b.$c`.

The following example adds indexes to the `Car` type, specifying three attributes as keys — `make`, `color`, and `year`:

```
Car.addIndex("$make");
Car.addIndex("$color");
Car.addIndex("$year");
```

If `Car` has subtypes, `addIndex()` will also add indexes to the subtypes. Later, if you add another subtype, JDD automatically adds an index to the subtype, using the same keys.

For most applications, the indexing provided by `addIndex()` is sufficient to optimize queries. However, if your application queries types that have many subtypes, you should consider adding superindexes; for more information, see Improving Query Performance with Superindexes on page 237.

After you have defined types and their attributes, you have provided the type information needed to create entities.

Creating Entities of a Type

Creating an accessible entity requires not just the construction of an entity, but also assigning values to its attributes and adding the entity to its type's extent. This last step makes the entity available to queries.

To create an entity, call the `create()` method on its type. This method invokes the `Entity()` constructor and returns an `Entity` object, as follows:

```
Entity car = null;
car = Car.create();
```

If you do not have a type object available for calling the `create()` method, you can call the static `findType()` method to get the type, as follows:

```
Type Car = Type.findType("Car", db);
```

Assigning attribute values

To assign attribute values in the entity, call the `put()` method on the attribute object, as follows:

```
make.put(car, carMake[i%4]);
color.put(car, carColor[i%3]);
year.put(car, 1980+i);
```

If you do not have the attribute object (for example, `make`), you can call `put()` on the entity (`car`), specifying the name of the attribute as an argument, as follows:

```
car.put("make", "Ford"); // allowed, but not recommended!
```

However, such calls can be expensive, especially when assigning values to the same attribute in a highly iterative loop. Calling an entity's `put()` method with the attribute name as an argument incurs the expense of a string lookup for each call.

Instead, call `put()` on the attribute object, specifying the entity object and the value to assign as arguments. If you do not have the attribute object, call the `findAttr()` method on the type to retrieve the attribute, as in the following example:

```
// make this call outside the access loop
StringAttribute make = (StringAttribute)Car.findAttr("make");

// make this call to assign the attribute value inside the loop
make.put(car, "Ford");
```

The same consideration applies when calling `get()` to retrieve an attribute value.

Adding an entity to the type's extent

The final stage in creating an entity is to add it to the extent of its type. Unless you add it to the extent, the entity is not put in the database. The following example adds a `car` entity to its type, `Car`:

```
Car.addToExtent(car);
```

At this point, you have populated the database and can perform queries.

Querying a Type

Queries are performed in JDD using the query mechanism provided by OSJI; for detailed information, see Querying PSE Pro Utility Collections. The only difference between OSJI queries and JDD queries is that, when you specify an attribute in a query string, you must precede the attribute name with the `$` character, just as you do when adding an index; see Defining Types and Their Attributes on page 227. You can cascade references through a hierarchy of attributes by preceding each attribute with the `$` character; for example, `$a.$b.$c`.

To query the database, construct a `TypeQuery` object, as follows:

```
TypeQuery query = new TypeQuery
  (Car, "($make == \"Fiat\") && ($year > 1990)");
```

You can use the `iterator()` method on the `TypeQuery` object to get a Java `Iterator` object and then use methods on the `Iterator` object to access the entities returned by the query.

A Simple JDD Application

The following sample program exercises the basic features of a JDD application to store and retrieve (through a query) information about cars.

The program creates a `Car` type and gives it three attributes: `make`, `color`, and `year`. The program then creates several `Car` entities, assigns values to their attributes, and queries the `Car` type for a list of entities that match the selection criteria specified by the query.

The command lines for compiling and executing the program follow the program listing.

```
// Cars.java: use the JDD classes to store information about
// cars; all type information is created at run time
import com.odi.*;
import com.odi.util.*;
import java.util.*;
import com.odi.jdd.*;
import com.odi.jdd.rel.*;

public class Cars {

  static Database db;
  static Transaction tr;

  static public void main(String args[]) {
    Session session = null;
    try {
      session = Session.create(null, null);
      session.join();
      load();
    } finally { session.terminate(); }
  }

  static void load() {
    String carMake[] =
      new String[] { "BMW", "Honda", "Edsel", "Fiat" };
    String carColor[]=new String[] { "red", "black", "green" };

    System.out.println("Creating database ...");
    try {
      Database.open("cars.db",
        ObjectStore.UPDATE).destroy();
    } catch (DatabaseNotFoundException e) { }

    db = Database.create("cars.db",
      ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);

    tr = Transaction.begin(ObjectStore.UPDATE);

    System.out.println("Generating type information ...");

    // create the type of the entities
    Type Car = new Type(db, "Car");

    // create attributes
    StringAttribute make = new StringAttribute(Car, "make");
    StringAttribute color = new StringAttribute(Car, "color");
```

```
    IntAttribute year = new IntAttribute(Car, "year");

    // add indexes to the attributes
    Car.addIndex("$make");
    Car.addIndex("$color");
    Car.addIndex("$year");

    System.out.println("Creating Entities ...");
    Entity car = null;
    for (int i = 0; i < 20; i++) {
      // create a Car entity
      car = Car.create();

      // assign values to its attributes
      make.put(car, carMake[i%4]);
      color.put(car, carColor[i%3]);
      year.put(car, 1980+i);

      // add the entity to its type's extent
      Car.addToExtent(car);
    }

    System.out.println(
      "Querying for all Fiats older than 1990 ...");
    TypeQuery query = new TypeQuery
      (Car, "($make == \"Fiat\") && ($year > 1990)");
    Iterator iter = query.iterator(null);
    System.out.println("\n  Make\tColor\tYear\n");
    while (iter.hasNext()) {
      car = (Entity)iter.next();
      System.out.println("  " + make.get(car) +
        "\t" + color.get(car) + "\t" + year.get(car));
    }
    tr.commit(); // done
  }
}
```

Here are the command lines to compile and execute the program, along with the output from the run:

```
C:\examples>javac Cars.java
C:\examples>java Cars
Creating database ...
Generating type information ...
Creating Entities ...
Querying for all Fiats older than 1990 ...

  Make      Color      Year

  Fiat      black      1999
  Fiat      green      1991
  Fiat      red      1995
```

Relationships

The `Relationship` class and its subtypes enable you to create and use bidirectional relationships as attributes. The relationship classes provide methods for adding related entities to the attributes as well as for accessing them.

The types of relationships and the classes that implement them are as follows:

- One-to-one, implemented by the `OneToOne` class
- One-to-many, implemented by the `OneToMany` class
- Many-to-many, implemented by the `ManyToMany` class

JDD provides automatic maintenance on relationships whenever a member object is added or removed.

The following sections describe the relationship types and how to use them in a JDD application.

One-to-One Relationships

The one-to-one relationship is implemented by the `OneToOne` class. The constructor for this class creates attributes on both sides of the relationship, each of which is represented by an entity reference to the other side. The signature for the constructor is

```
public OneToOne(Type OneSide,
  java.lang.String OneSideAttribute,
  Type OtherSide,
  java.lang.String OtherSideAttribute)
```

You can use the constructed `OneToOne` object as an attribute object. For example, to assign related entities to the relationship, you would invoke the `put()` method on the `OneToOne` object. The signature of `put()` is

```
void put(Entity OneSideEntity, Entity OtherSideEntity)
```

To access the entity in the attribute, call the `get()` method on the attribute. Invoking the `OneSideAttribute.get()` method returns `OtherSideEntity`, and invoking the `OtherSideAttribute.get()` method returns `OneSideEntity`.

One-to-Many Relationships

The one-to-many relationship is implemented by the abstract class `OneToMany`. A one-to-many relationship has an owner (the “one” side) and members (the “many” side).

To create a one-to-many relationship, use the constructor for one of the subclasses, `LinkedOneToMany` or `IndexedOneToMany`. Their signatures are as follows:

```
public LinkedOneToMany(Type ownerType,
  java.lang.String ownerAttr,
  Type memberType,
  java.lang.String memberAttr
  boolean doublyLinked)
```

```
public IndexedOneToMany(Type ownerType,
  java.lang.String ownerAttr,
  Type memberType,
  java.lang.String memberAttr
  java.lang.String primaryIndexPath)
```

Both constructors create attributes for the types *ownerType* and *memberType*. The attribute on the owner side is a set of entity references to *memberType* entities, and the attribute on the member side is an entity reference to an *ownerType* entity. For both constructors, the constructed object can be used as an *ownerAttr* object.

The constructor signatures for `LinkedOneToMany` and `IndexedOneToMany` differ in their last argument, as follows:

- The *doublyLinked* argument to `LinkedOneToMany()` specifies the type of list to use for the member entity references. If the *doublyLinked* argument is `true`, the list is doubly linked; otherwise, it is singly linked. A doubly linked list is more efficient when removing elements.
- The `IndexedOneToMany()` constructor implements the *ownerAttr* attribute as an `OSTreeSet`. It is for use with a very large set that you expect to query. The *primaryIndexPath* argument specifies the primary index; if this argument is omitted, no primary index is created. For more information about `OSTreeSet`, see Description of OSTreeSet on page 143.

The sample program listed in Relationship Example on page 235 uses the `LinkedOneToMany()` constructor to implement a one-to-many relationship between `Book` entities and `Borrower` entities, as follows:

```
LinkedOneToMany books = new LinkedOneToMany
  (Borrower, "books", Book, "BorrowedBy", true);
```

In the next line, the constructed object (`books`) is used to call the `add()` method, which adds the `Book` and `Borrower` entities to the relationship:

```
books.add(borrower[i%2], book);
```

The `add()` method returns `false` if the entities were already related in this relationship; otherwise, it returns `true` and adds them to the relationship.

The next line retrieves the borrower of a book, by calling the `get()` method on `book`, specifying the name of the attribute (`BorrowedBy`) as an argument:

```
Entity person = (Entity)book.get("BorrowedBy");
```

The next line makes two calls. The first call is to invoke the `get()` method on `books` (an attribute of the `Borrower` type), which returns a `ToManySet` object. This object is the set of `Book` objects related to `person`. The `iterator()` method is then invoked on the `ToManySet` object and returns an `Iterator` object that can be used to access the list of borrowed books:

```
Iterator iter = books.get(person).iterator();
```

Many-to-Many Relationships

The many-to-many relationship is represented by the abstract `ManyToMany` class. An object of this class maintains sets of entities on both sides of the relationship. Whenever an element is added to the set that represents one side of the relationship, JDD maintains the inverse direction as well. Similar relationship maintenance is performed when removing elements from a set.

To create a many-to-many relationship, call the `ManyToMany()` constructor. Its signature is as follows:

```
public ManyToMany(Type ownerType,
  java.lang.String ownerName,
  Type memberType,
  java.lang.String memberName)
```

To add related entities to the relationship, call the `add()` method on the `ManyToMany` object. Its signature is as follows:

```
public boolean add(Entity ownerEntity, Entity memberEntity)
```

The `add()` method returns `false` if *`ownerEntity`* and *`memberEntity`* were previously linked in this relationship; otherwise, it adds entity references to the attributes and returns `true`.

Linked Objects and Many-to-Many Relationships

Some applications require a many-to-many relationship that can also represent the linking of two entities in a *link object*. The `ManyToManyWithObject` class can be used to implement both the relationship as well as link objects.

Consider, for example, an application that tracks students and the courses in which they are enrolled. The application creates two types, `Student` and `Course`. It implements a many-to-many relationship so that a `Course` entity can reference all the students taking the course, and a `Student` entity can reference all courses for which a student is enrolled. Furthermore, the application uses a link object to represent each student's enrollment in a course. The link object has a type (for example, `Enrollment`), enabling it to have attributes (for example, `grade`).

The `ManyToManyWithObject()` constructor has the following signature:

```
public ManyToManyWithObject(Type ownerType,
  java.lang.String ownerName,
  Type memberType,
  java.lang.String memberName,
  java.lang.String linkObjectType)
```

The *`linkObjectType`* argument is the type of the link object. If a type with that name already exists, JDD uses the existing type; otherwise, it creates a new type.

The constructor implements the relationship as two maps, one on each side of the relationship. The keys of each map are the related entities, and the values are the link objects. To illustrate with the previous example, each `Student` entity can have a map keyed by `Course` entities with values of `Enrollment` objects. And each `Course` entity can have a map keyed by `Student` entities, whose values are also `Enrollment` objects.

Accessor methods can be used to do the following:

- Retrieve the map of link objects.
- Get the set of entities referenced in the attribute.
- Add related entities to the relationship and create a link object.
- Unlink or remove entities from the map and delete the link object.

Relationship Example

The example program listed in this section is a simple lending- library application that keeps a record of borrowers and the books each has borrowed. It creates two types, `Book` and `Borrower`, and uses the `LinkedOneToMany` class to implement a one-to-many relationship that represents the relationship between borrower and book. A borrower can have many books, but a book can have only one borrower.

The `Book` and `Borrower` types each have two attributes. The `Book` type has these attributes:

- `title`, the title of a `Book` entity
- `BorrowedBy`, a reference to a `Borrower` entity

The `Borrower` type has these attributes:

- `name`, the name of a `Borrower` entity
- `books`, the list of the `Book` entities loaned out to `Borrower`

The `title` and `name` attributes are created by invoking the `StringAttribute()` constructor. The `BorrowedBy` and `books` attributes are created by invoking the `LinkedOneToMany()` constructor.

Each of the `Book` entities is created in a `for` loop. After an entity is created, the book it represents is recorded as having been loaned out to a borrower by adding the related `Book` and `Borrower` entities to the one-to-many relationship.

The command lines for compiling and executing the program are shown following the source code listing.

```
// Library.java: use JDD classes to store and retrieve
// information for a lending library -- who has borrowed which
// books. The one-to-many relationship between borrower and
// books is implemented by JDD's LinkedOneToMany class.
import com.odi.*;
import com.odi.util.*;
import java.util.*;
import com.odi.jdd.*;
import com.odi.jdd.rel.*;

public class Library {

  static Database db;
  static Transaction tr;

  static public void main(String args[]) {
    Session session = null;
    try {
```

```
    session = Session.create(null, null);
    session.join();
    loan();
  } finally { session.terminate(); }
}

static void loan() {
  Entity borrower [] = new Entity[2]; // borrowers

  String titleStr[] = new String[] { // titles of books
    "Life of Johnson", "Tale of a Tub", "Rambler",
    "Beggar's Opera", "Memoirs of Martinus Scriblerus",
    "Reflections", "Seventeen Thirty-Eight"
  };

  // create database
  try {
    Database.open
      ("library.db", ObjectStore.UPDATE).destroy();
  } catch (DatabaseNotFoundException e) { }

  db = Database.create("library.db",
    ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);

  tr = Transaction.begin(ObjectStore.UPDATE);

  // create Book type with title attribute
  Type Book = new Type(db, "Book");
  StringAttribute title = new StringAttribute(Book,"title");

  // create Borrower type with name attribute
  Type Borrower = new Type(db, "Borrower");
  StringAttribute name =
    new StringAttribute(Borrower, "name");

  // create a one-to-many relationship between Borrower and
  // Book
  LinkedOneToMany books = new LinkedOneToMany
    (Borrower, "books", Book, "BorrowedBy", true);

  // create two Borrower entities
  borrower[0] = Borrower.create();
  borrower[0].put(name, "Terry");
  Borrower.addToExtent(borrower[0]);
  borrower[1] = Borrower.create();
  borrower[1].put(name, "Alice");
  Borrower.addToExtent(borrower[1]);

  // create Book entities
  Entity book = null;
  for (int i = 0; i < 7; i++) {
    book = Book.create();
    title.put(book, titleStr[i]);
    Book.addToExtent(book);

    // lend them out to the borrowers
    books.add(borrower[i%2], book);
  }

  // Who borrowed the last book?
```

```
    Entity person = (Entity)book.get("BorrowedBy");
    System.out.println("\"" + title.get(book) +
      "\" is out on loan to " + person.get(name));

    // get the list of books loaned to this borrower
    Iterator iter = books.get(person).iterator();
    System.out.println(person.get(name) +
      " borrowed these books:");
    while (iter.hasNext())
      System.out.println("\t" + ((Entity)iter.next()).get(title));

    tr.commit(); // done
  }
}
```

Here are the command lines to compile and execute the program, along with the output from the run:

```
C:\examples>javac Library.java
C:\examples>java Library
"Seventeen Thirty-Eight" is out on loan to Terry
Terry borrowed these books:
  Life of Johnson
  Rambler
  Memoirs of Martinus Scriblerus
  Seventeen Thirty-Eight
```

Improving Query Performance with Superindexes

A superindex is a special type of index for optimizing queries on types that have many subtypes. The following sections provide background information to explain how JDD implements default indexing; when default indexing may not be sufficient to optimize queries; and how superindexes can improve the performance of queries.

Note

Applications that do not query types with many subtypes will probably not benefit from superindexing and should use the default indexing as described in Defining Types and Their Attributes on page 227.

Queries and Default Indexing

When an application queries a type, by default the query is recursive: separate query operations execute on the type and on any of its subtypes. Likewise, when you call `addIndex()` on a type, the index is recursively added to the type and to all its subtypes. These indexes are automatically maintained and updated by JDD when you add or remove a subtype, or when you change the value of an attribute.

This default level of indexing is sufficient for most query operations. However, if you query a parent type that has a large and intricate hierarchy of subtypes, indexing can add to the overhead of the query. To reduce the time it takes to query such types, JDD provides the superindex.

Superindexing

When you call `addSuperIndex()` on a type, JDD adds a single superindex to the type, which indexes the parent type and all its subtypes. When you query the parent type, only one query operation occurs — not the recursive queries that occur when you query a type with default indexing. As with default indexing, JDD automatically updates the superindex if you add or remove a subtype or if you change the value of an attribute.

There are several disadvantages to using a superindex that you should consider when deciding which type of index to add:

- A superindex can benefit query performance only when added to types having many subtypes. If your application queries types with few or no subtypes, adding a superindex will not significantly improve query performance.
- Invoking the `addSuperIndex()` method creates a single superindex, which is added to the type object to which the method is applied. If you begin the query with a subtype, you do not get the benefit of *any* indexing — unless you have explicitly called `addSuperIndex()` on the subtype.
- When you change an attribute value for a type that has a superindex, JDD automatically updates any superindexes that have been added to the subtypes. If you have added many superindexes to many subtypes, index maintenance can become time consuming.
- Superindexes may not be added to types that have subtypes in different segments of the same database or in different databases. Once a superindex has been added to a type, no new subtypes may be added that are in a different database or segment.

Mixing Java Objects with JDD

JDD enables you to mix persistent Java objects and JDD entities in the same application. JDD classes are extendable. For example, you can extend the `Entity` class to have native Java fields and methods. You can similarly extend the `Type` and `Attribute` classes.

When you extend JDD classes to include native fields, however, you lose some of the flexibility that JDD provides. The reason is that, to make the extended classes persistence-capable, you must use the PSE Pro API to do the following:

- Run the `osjcfp` postprocessor to make the classes persistence capable.
- Perform schema evolution if you change any of the persistence-capable classes.

If you only have to run the postprocessor once and never have to perform schema evolution, these tasks may not be an issue. But if your application models data that is constantly changing, you will probably want to rely as much as possible on JDD for persistent storage.

This section is organized as follows:

When You Must Use the PSE Pro API

You must use the PSE Pro API to perform the following operations:

- Creating and managing sessions; for more information, see How Sessions Keep Threads Organized.
- Opening and closing a database; for more information, see Chapter 4, Managing Databases.
- Starting and committing a transaction; for more information, see Chapter 5, Working with Transactions.

Pros and Cons of Mixing the PSE Pro API with JDD

Among the reasons for using the PSE Pro API to include persistence-capable classes in a JDD application are the following:

- Native fields of Java objects take up much less space in the database than JDD attributes.
- Accessing and modifying native fields is significantly faster.

If either of these factors weighs heavily with your application, you might consider extending JDD classes to define native fields and accessor methods.

Fields vs. attributes

When deciding whether to store data as a native field or an attribute, you should consider whether you will have to change the field at a later date. Deleting or renaming a field or changing its type requires recompiling source code, running the postprocessor, and possibly performing schema evolution. Changing an attribute, on the other hand, has no such risks.

Access time

Another factor to consider is access time: accessing a native field is significantly faster than accessing an attribute. If you need to access the same data in a highly iterative, time-critical loop, you should consider storing it as a native field.

Using Extended JDD Classes

The most likely candidate for extending is the `Entity` class. After compiling and postprocessing an application that uses the extended entity, you use the application to create a type for the entity, give the type a set of attributes, and create entities of the type — using the same procedure described in Defining Types and Their Attributes on page 227 and Creating Entities of a Type on page 228.

Creating attribute objects for native fields

If you want to use any of the JDD accessor methods on a native field, you must first create an attribute object for the field, as described in Creating Entities of a Type on page 228. Note, however, that using a JDD method to access a native field is significantly slower than using a native method. When you use a JDD accessor method on an attribute object associated with a native field, JDD uses Java reflection to access the field.

Querying native fields

JDD allows you to mix native fields and attributes in the same query string. The query treats the field reference differently, depending on whether or not you precede the name with the $ character:

- If the name begins with the $ character (for example, `$age`), JDD interprets it as the name of an attribute object. If the object is associated with a native field, the query uses Java reflection to access the field.
- If the name does *not* begin with the $ character (for example, `age`), JDD interprets it as the name of a native field. The query accesses the field directly, just as an PSE Pro query would.

Indexing native fields

Creating attribute objects for native fields also allows you to add indexes to the fields. JDD performs the same index maintenance on native fields as on attributes, *except* if either of the following is true:

- The PSE Pro API was used to add the index.
- A field was updated directly instead of through a JDD accessor method.

If either condition is true, JDD does not update the index.

Example

The following example uses a persistence-capable class, `UserEntity`, which extends JDD's `Entity` class and defines a native integer field called `age`:

```
// create the type User in the database represented by db
Type User = new Type(db, "User");

// create an entity of the type User; UserEntity extends Entity
UserEntity user = new UserEntity(User);

// assign to the age field
user.age = 32; // native Java field
```

To use a JDD method to access `age`, you first create an attribute object for it, as follows:

```
// create an attribute object for the age field, which is
// defined in the UserEntity class
IntAttribute ageAttr =
        (IntAttribute)myUser.findAttr("age");

// call JDD's put() method on the attribute object for the age // field
ageAttr.put(user, 32);
```

Chapter 11
Using Java Data Objects (JDO) with PSE Pro

PSE Pro for Java provides a complete implementation of the Java Data Objects (JDO) 1.0.1 specification. Creating your application with JDO is an alternative to using the native PSE Pro interface for storing persistent data. This chapter describes how to configure PSE Pro applications to use JDO. The chapter contains the following sections:

Note

PSE Pro applications should not mix data created by the JDO interface and data created by the native ObjectStorePSE Pro interface. Because the two interfaces have different models of persistence support, applications for a given ObjectStorePSE Pro database should consistently use only one interface. Progress Software Corporation makes no claim regarding the compatibility of the two interfaces.

You need JDK 1.5 or later to use JDO with PSE Pro.

PSEJDO interface is implemented to be compliant with the 1.0.1 Specification. It is recommended that the user become familiar with the following sites for JDO related API and information:

- `http://java.sun.com/products/jdo`
- `http://www.JDOcentral.com`
- `http://groups.yahoo.com/group/JavaDataObjects`

Overview of JDO with PSE Pro

After defining `PersistenceCapable` classes (classes whose instances can be stored in a database), developers need to write very little additional code to allow their applications to use persistent data.

The ObjectStorePSE storage engine in conjunction with the PSEJDO interface allows you to quickly read or modify portions of your persistent data. You are not required to read in all persistent data when you just want to look at a subset. This reduces

start-up and transaction commit times and allows you to run much larger Java applications without increasing the amount of memory or swap space on the system.

When you access persistent data inside a transaction, PSEJDO ensures that your results are not compromised by other users sharing the data. If something goes wrong, or if you determine that you do not want to keep changes, you can abort the transaction. In that case, ObjectStorePSE Pro restores the database to the state it was in before the transaction started. This makes it straightforward to write applications that have to recover from exceptions or failures.

PSEJDO is designed for applications that need standard persistent support and utilize a single process architecture. The implementation is 100% pure Java, uses less than 600 KB of disk space, and runs entirely within the application process. It supports transactions and provides access to objects without reading the entire database.

What does PSEJDO do?

The JDO for PSE Pro product provides the user a standard way to realize transparent persistence with a robust storage engine. The combination of the JDO implementation and PSE Pro storage engine allows the user to use:

- Multi-threaded applications to use data in the database
- Start, commit and abort transactions to access data in the database
- Automatic extent maintenance to help manage locating objects
- Store objects in a database, retrieve and update those objects

PSEJDO can recover from an application failure or system crash. If a failure prevents some of the changes in a transaction from being saved to disk, the product ensures that none of that transaction's changes are saved in the database. When you restart the application, the database is consistent with the way it was before the transaction started.

PSE Pro does not have facilities for protection against media failure. The full ObjectStore product provides a feature set than can help users construct solutions to deal with this kind of failure; see `www.objectstore.net` for more information.

JDO for PSE Pro can support

- Hundreds of thousands of objects in a database and hundreds of megabytes of data in a database
- Multiple `PersistenceManagers` per database
- An ability to collect garbage in a database
- JDO Collection and Extent support
- JDO Query support
- Full database recovery from system failure
- Utilities for displaying information about the objects in the database and checking the references between objects

Description of PSEJDO Architecture

PSEJDO is a Java library that runs entirely within your Java virtual machine process. A PSE Pro database consists of three files. PSE Pro uses the standard Java class `java.io.RandomAccessFile` to access a database. There is no client/server distinction.

There is no PSE Pro server. PSE Pro accesses a file on the file system. PSE Pro has no special privileges. If PSE Pro can read or write a file, so can any other user using ordinary file I/O.

PSEJDO does not lock objects in memory. It uses weak references to maintain objects in memory, allowing the java garbage collector to work as expected to keep virtual memory available.

Because PSEJDO is compliant to a standard interface, all the key interfaces are defined by the JDO standard. In this section, the major interfaces are highlighted to give background before examining the sample code.

PersistenceCapable

An interface implemented by all user classes that can be stored in the database. This interface creates the contract that a `PersistenceManager` uses to manage the state of a persistent object. User classes that need to go into the database will implement this interface. There is an *enhancement* process that is used to alleviate the need for the developer to write any code to comply to this interface.

PersistenceManager

Instances of this interface are used to create context for accessing persistent data. PSEJDO provides the classes that implement this interface. Developers utilize this to inquire or cause effect on lifecycle actions of `PersistenceCapable` classes. Developers use this interface to create transactions to access persistent data.

PersistenceManagerFactory

This interface is used to create `PersistenceManager` instances. PSEJDO provides the classes that implement this interface. OSJDO uses a `Properties` object to get a `PersistenceManagerFactory` instance . See PSEJDO Properties File on page 249 and Standard JDO Properties on page 249 for more information about these properties. The `PersistenceManagerFactory` uses these properties to configure `PersistenceManager` instances.

JDOHelper

This is used to bootstrap the PSEJDO implementation. The class provides a series of management and administration functions. This is typically used to create an instance of a class that implements `PersistenceManagerFactory`.

Extent

A class that implements `Extent` represents a collection of all objects of a particular type. All classes that are `PersistenceCapable` will have an `Extent` by default. You can specify if classes should not have an `Extent` so as to avoid unnecessary database

overhead. JDO expects to use `Extents` to create the initial navigational access into a database.

Query

Classes that implement the `Query` interface are instantiated from a `PersistenceManager`. A `Query` can be used to apply a restriction across `Collections` and/or `Extents` to return qualified objects.

Query Limitations

If a `contains` expression is used in the query filter, the argument to the `contains` method should be `Decalared Variable`. Also, the `contains` expression should be a left operand of an `AND` expression and `Declared Variable` should be used in the right operand of that `AND` expression (see the third paragraph of section 14.6.5 of the JDO 1.0.1 specification).

Navigation is not currently supported in the ordering specification. This implies that the query results can currently be sorted only on the sortable fields of the result type. For example, if a query execution returns a result set that contains elements of the type `Person` and `Person` is defined as:

```
public class Person {
  String name;
  int age;
  Person spouse;
}
```

the query results can be sorted based on either `name` or `age`, but not `spouse.name` or `spouse.age`.

Query Enhancements

You can use public method invocations in a query filter. However, you cannot use invocations of methods named `contains` because that is a key word in the JDOQL.

You can use pattern matching expressions via the pattern matching operator "~~" as in native PSE Pro queries.

The PSE Pro implementation of queries generates classes dynamically and loads them using the PSE Pro Class loader. This may cause problems in applications running under a security manager if the security policy does not take this situation into account. Applications running under a security manager need to take this into account when specifying security policies for the application.

Creating JDO Applications

In order to use the PSEJDO interface, your application must include the standard JDO components as implemented by PSE Pro. These components include:

- `PersistenceManagerFactory`

  You create a `PersistenceManagerFactory` by using `JDOHelper.getPersistenceManagerFactory()` setting various runtime configuration properties

- `PersistenceManager`

  You acquire a `PersistenceManager` by invoking `PersistenceManagerFactory.getPersistenceManager()`. The `PersistenceManager` will have the characteristics as defined by the properties used to create the `PersistenceManagerFactory`.

- `Transaction`

  Transactions are used to define units of atomic work. You create a `Transaction` by invoking `PeristenceManager.currentTransaction()`.

  Start and commit a transaction by calling `Transaction.begin()` and `Transaction.commit()`.

- `PersistenceCapable`

  `PersistenceCapable` objects are stored in the database in two ways. The first is to make a object persistent with a call to `PersistenceManager.makePersistent()` for a single instance or with a call to `PersistenceManager.makePersistentAll()`for multiple instances. The second way an object becomes persisistent is if it is reachable from an object that was made persistent.

  When a `PersistenceCapable` object becomes persistent it is automatically added to the `Extent` for its class unless the class is specified to not require an `Extent`.

  Persistent objects are retrieved and displayed by accessing the `Extent` for the class.

- PSEJDO enhancer

  When you write a PSEJDO program for PSE Pro, you write it as though your user classes already implement `PersistenceCapable`. However, a program cannot store objects persistently until you run the PSEJDO enhancer on your compiled class files. The enhancer generates an annotated version of the class files. The annotated version of the class definition is persistence capable. You run the enhancer after you compile the program and before you run the program.

Developing Applications

Before you can run applications that use the JDO interface, you must

- Add relevant PSE Pro and JDO files to your `CLASSPATH` environment variable.
- Create the persistence descriptor file.
- Compile the source file.
- Run the JDO enhancer on the `.class` files.

Adding Files to Your CLASSPATH

In your `CLASSPATH` environment variable, you need files for the Standard JDO interfaces, PSEJDO implementation files, and PSE Pro implementation files. All the files are located in the `lib` directory of the PSE Pro installation.

- `jdo.jar` — The standard JDO interface
- `psejdo.jar` — The PSEJDO implementation
- `pro.jar` — Standard PSE Pro for Java
- `tools.jar` — Standard PSE Pro for Java

Make sure all `jar` files are explicitly in your classpath; an entry for the directory that contains them is not sufficient.

For example, the default installation directory for PSE Pro on Windows is `C:\ODI\PSEProJ` and you need the following addition to your `CLASSPATH`:

```
C:\ODI\PSEProJ\lib\pro.jar;
  C:\ODI\PSEProJ\lib\tools.jar;
  C:\ODI\PSEProJ\lib\jdo.jar;
  C:\ODI\PSEProJ\lib\psejdo.jar;
  C:\ODI\PSEProJ
```

The default installation directory on UNIX,is `/usr/local/odi/pseproj` and you need the following addition to your `CLASSPATH`:

```
/usr/local/odi/pseproj/lib/pro.jar:
  /usr/local/odi/pseproj/lib/tools.jar:
  /usr/local/odi/pseproj/lib/jdo.jar:
  /usr/local/odi/pseproj/lib/psejdo.jar:
  /usr/local/odi/pseproj
```

Persistence Descriptor

This is an XML file specifing the metadata that describes the classes that are to be stored persistently. It is used at both enhancement time and at runtime. It also can contain particular information that may influence the enhancement or runtime process. Some of the tags are defined by the JDO standard. It is also possible for vendors to augment the XML by using the `<extension>` tag to specifically optimize or configure a particular product. PSE Pro does not currently add vendor-specific extensions.

The JDO enhancer uses the `CLASSPATH` to resolve the location of the Document Type Definition (DTD). A DTD compliant with the 1.0.1 JDO specification is included in the `jdo.jar` file, so the JDO enhancer will automatically find it.

The name for the descriptor file follows the JDO specification; if it describes a single class, it should be named *`class-name`*`.jdo` and if it describes a package it should be named `package.jdo`.

The key tags in the persistence descriptor file are as follows:

`<jdo>` - namespace for jdo

`<package>` - java package name for classes to be enhanced

`<class>` - The Java class that needs to be annotated. The attribute `<identity-type>` should always be `datastore` and the attribute `<requires-extent>` can be `true` or `false`.

`<field>` - Fields of the class that should be stored persistently. The attribute `<persistence-modifier>` can be `persistent`, `transactional`, or `none`. PSEJDO always treats the attribute `<default-fetch-group>` as though it is set to `true`.

For an example XML persistence descriptor file, refer to Example Persistence Descriptor on page 258.

Compiling the Program

You compile PSE Pro applications that use the JDO interface just as you would any other PSE Pro application. For example, to compile the example program at the end of this chapter, change to the PSE Pro installation directory and, on Windows platforms, enter

```
javac com\odi\demo\jdo\people\*.java
```

On UNIX platforms, enter

```
javac com/odi/demo/jdo/people/*.java
```

As output, the `javac` compiler produces the byte code class file `Person.class` in the same directory that contains the `Person.java` file

Running the JDO Enhancer

For PSE Pro applications that use the JDO interface, you annotate the persistent classes with the JDO enhancer. You use the JDO enhancer instead of the PSE Pro postprocessor that is used to annotate classes for the native PSE Pro interface.

After you compile your application, you run the JDO enhancer on the classes you want to store in your database. The JDO enhancer modifies these classes so that they implement the `PersistenceCapable` interface. In addition, the JDO enhancer annotates the class methods so they can manage the lifecycle of the persistent classes in the JVM. The enhancer generates new annotated class files. After you run the enhancer, you should use the annotated class files rather than the original class files.

The command to run the JDO enhancer is

```
osenhance -dest <destination_directory> [-cpath <classpath>]
```

```
args [options]
```

args
This specifies the classes you want to annotate. You can use the class names of the classes or the file names, for example `com.odi.demo.class1` or `/build/classes/com/odi/demo/class1.class`. The associated persistence descriptor XML file must be available on the JDO enhancer classpath. You need to include all classes you want to make persistent.

options
You can use the following options to `osenhance`:

`{ -cpath | -classpath } <classpath>`
Specifies the path by which to locate class files for enhancing. If you specify this option, PSE Pro uses it in place of the `CLASSPATH` environment variable. The `-cpath` option does not affect the class path used to run the enhancer.

`{ -d | -dest } <destination_directory>`
This option is required. The JDO enhancer uses the directory you specify for *`destination_directory`* as the root for locating the annotated files. The JDO enhancer places each class file it operates on in the package-appropriate subdirectory of the destination directory, as though the destination directory were in your class path.

If the destination directory specification would cause the enhancer to overwrite an original file and you did not specify the `-inplace` option, the enhancer reports an error and terminates without producing any output.

`{ -f | -force }`
Forces the postprocessor to overwrite existing annotated `.class` files.

`-inplace`
Causes the JDO enhancer to annotate stand-alone files — files that are not in `.zip` files or `.jar` files — in place rather than writing the annotated file in the destination directory. When the enhancer annotates a class in place, it overwrites the original class files with the annotated class files. If a class originates in a `.zip` file or `.jar` file, the enhancer writes the annotated class to the destination directory.

Do not use this option when you are doing iterative development. During development, a separate output directory avoids errors and supports debugging.

When you specify the `-inplace` option, you must still specify a destination directory, but the enhancer ignores it for stand-alone files.

`-nowrite`
Performs process and error checking but does not actually annotate class files. This option allows a test run of the JDO enhancer. You use it to determine whether or not all specified classes are accessible, whether additional options are needed, and if you specify `-v` (verbose), you can see where the resulting files would be located.

`{ -q | -quiet }`
Causes the JDO enhancer to refrain from displaying warnings. A warning message provides information about something that the JDO enhancer recognizes

as a possible problem but cannot confirm as actually being a problem. This option cancels a previous `-verbose` option, if you specified one.

`{ -v | -verbose }`
Causes the JDO enhancer to write descriptions of its actions to standard output. This option cancels a previous `-quiet` option, if you specified one.

PSEJDO Properties File

PSEJDO uses a `Properties` object that contains the properties necessary for setting up the `PersistenceManagerFactory` class. The properties should include relevant standard JDO properties as well PSEJDO vendor-specific properties. The information for these properties can be loaded from a file. For example, the example application at the end of this chapter uses a file called `jdo.properties`. It contains the following:

```
javax.jdo.PersistenceManagerFactoryClass=
      com.odi.jdo.OsjdoPersistenceManagerFactory
javax.jdo.option.ConnectionURL=
      osjdo:C:\ODI\PSEProJ\com\odi\demo\jdo\people\person.odb
com.odi.jdo.option.ConnectionOpenMode=update
com.odi.jdo.option.Product=PSEPro
com.odi.jdo.option.ConnectionOpenAction=auto
```

Standard JDO Properties

To obtain a `PersistenceManagerFactory`, a JDO application typically sets up a `Properties` object containing the desired set of properties, and passes it to `JDOHelper.getPersistenceManagerFactory`. The standard JDO properties are described in this section.

javax.jdo.PersistenceManagerFactoryClass

The value of the `PersistenceManagerFactoryClass` property determines the type of `PersistenceManagerFactory` object returned by `JDOHelper.getPersistenceManagerFactory`. OSJDOPSEJDO applications should set the value of this property to `com.odi.jdo.OsjdoPersistenceManagerFactory`.

javax.jdo.option.ConnectionURL

In PSEJDO, this property specifies the name of the PSE Pro database that will provide context for operations such as `makePersistent` and `getExtent`.

The syntax of the `ConnectionURL` property given when acquiring an `OsjdoPersistenceManager` is as follows: a scheme specifier of `osjdo` followed by a colon, followed by a PSE Pro database pathname, for example `osjdo:/usr/guy/Person.odb`.

This property can also be manipulated by the following methods:

- `String PersistenceManagerFactory.getConnectionURL()`
- `void PersistenceManagerFactory.setConnectionURL(String v)`

javax.jdo.option.Optimistic

This boolean property specifies that generated `PersistenceManagers` will use optimistic transactions, an optional feature of JDO that is not currently supported by PSE Pro.

javax.jdo.option.RetainValues

This `boolean` property controls the treatment of persistent instances after commit. If `true`, then the instance is retained in the cache between transactions as an accessible persistent-nontransactional object. Otherwise the instance cannot be read between transactions. Defaults to `false`.

This property can also be manipulated by the following methods:

- `boolean PersistenceManagerFactory.getRetainValues()`
- `void PersistenceManagerFactory.setRetainValues(boolean v)`

javax.jdo.option.RestoreValues

This `boolean` property controls the treatment of persistent-new instances after rollback. If `true`, then an instance's fields are restored to their values as of the time the instance became persistent-new. Defaults to `false`.

This property can also be manipulated by the following methods:

- `boolean PersistenceManagerFactory.getRestoreValues()`
- `void PersistenceManagerFactory.setRestoreValues(boolean v)`

javax.jdo.option.IgnoreCache

This property has no effect in PSEJDO.

javax.jdo.option.NontransactionalRead

Setting this option to `true` allows reading of persistent data when there is no transaction in progress. Defaults to `false`.

javax.jdo.option.NontransactionalWrite

This `boolean` property enables an optional feature that is not implemented in PSEJDO. Attempts to set it result in `JDOUnsupportedOptionException`.

javax.jdo.option.Multithreaded

This `boolean` property indicates to the JDO implementation whether the application intends to allow multiple threads to access a `PersistenceManager`. Defaults to `false`.

This property can also be manipulated by the following methods:

- `boolean PersistenceManagerFactory.getMultithreaded()`
- `void PersistenceManagerFactory.setMultithreaded(boolean v)`

javax.jdo.option.ConnectionUserName
javax.jdo.option.ConnectionPassword

These two `String`-valued properties specify a user name and password for the database connection. In PSE Pro, these options have no effect.

These properties can also be manipulated by the following methods:

- `String PersistenceManagerFactory.getConnectionUserName()`
- `void PersistenceManagerFactory.setConnectionUserName(String v)`
- `void PersistenceManagerFactory.setConnectionPassword(String v)`

javax.jdo.option.ConnectionFactoryName
javax.jdo.option.ConnectionFactory2Name

These two properties are currently not used in PSEJDO, and attempts to set them are ignored.

PSEJDO Specific Properties

This section describes the PSEJDO vendor-specific properties.

com.odi.jdo.option.ConnectionOpenAction

This `String` valued property indicates whether the database named in `ConnectionURL` should be created or merely opened. Possible values are `create`, `open`, and `auto`. If the value is `create`, the database will be created, and any existing database with the same name will be destroyed. If the value is `open`, the implementation will expect the database to be already extant and open it in the mode indicated by the `ConnectionOpenMode` property. If the value is `auto`, then the database will be created if it does not already exist, and opened in update mode if it does not exist. The default value is `open`.

This property can also be manipulated by the following methods:

- `String OsjdoHelper.getConnectionOpenAction( PersistenceManagerFactory pmf )`
- `void OsjdoHelper.setConnectionOpenAction( PersistenceManagerFactory pmf, String v )`

com.odi.jdo.option.ConnectionOpenMode

This `String` valued property indicates whether the database named in ConnectionURL should be opened in update or read-only mode. Possible values are the Strings `update`, `mvcc`, and `readonly`. Defaults to `update`. Attempts to give this property values not mentioned here result in a `JDOFatalUserException` being thrown. The value of `ConnectionMode` is ignored unless `ConnectionOpenAction` is set to `open`.

This property can also be manipulated by the following methods:

- `Boolean OsjdoHelper.getConnectionOpenMode( PersistenceManagerFactory pmf )`
- `String OsjdoHelper.setConnectionOpenMode( PersistenceManagerFactory pmf, String v )`

com.odi.jdo.option.Product

This property is mapped to the PSE Pro `Session` property `com.odi.product`. By this means the application can choose the underlying implementation to use for storing JDO persistent objects. For PSE Pro, the value of this option should be `PSE Pro`. However, if PSE Pro, is the only implementation available, there is no need to set this option. It is needed only when both PSE Pro and ObjectStore are available in the runtime environment.

This property can also be manipulated by the following methods:

- `String OsjdoHelper.getProduct(PersistenceManagerFactory pmf)`
- `void OsjdoHelper.setProduct(PersistenceManagerFactory pmf, String v)`

com.odi.jdo.option.StringPoolSize

This property is mapped to the PSE Pro, `Session` property `com.odi.stringPoolSize`.

This property can also be manipulated by the following methods:

- `int OsjdoHelper.getStringPoolSize(PersistenceManagerFactor pmf)`
- `void OsjdoHelper.setStringPoolSize(PersistenceManagerFactory pmf, int v)`

PSEJDO Feature Set

The 1.0.1 JDO specification describes many optional features. The table below lists the JDO features and whether or not they are available with PSEJDO.

The JDO interface provides an API to examine the optional supported features. An example program is provided in the `com\odi\demo\jdo\features` directory. The list returned when you run the program indicates the features that are supported. You can run this program by running the following commands:

1 On Windows

```
javac C:\ODI\PSEProJ\com\odi\demo\jdo\features\*.java
```

On Unix:

```
javac /opt/ODI/PSEProJ/com/odi/demo/jdo/features/*.java
```

2 On either platform run:

```
java -Djavax.jdo.PersistentManagerFactoryClass=
      com.odi.jdo.OsjdoPersistentManagerFactory
      com.odi.jdo.demo.feature.OptionalSupport
```

Optional JDO Feature	*PSEPRO Support*
`javax.jdo.option.TransientTransactional`	Yes, except for fields declared transactional in the meta data
`javax.jdo.option.NontransactionalRead`	Yes

javax.jdo.option.NontransactionalWrite	No
javax.jdo.option.RetainValues	Yes
javax.jdo.option.RestoreValues	Yes
javax.jdo.option.Optimistic	No
javax.jdo.option.ApplicationIdentity	No
javax.jdo.option.DataStoreIdentity	Yes
javax.jdo.option.NondurableIdentity	No
javax.jdo.option.ArrayList	No
javax.jdo.option.HashMap	No
javax.jdo.option.Hashtable	No
javax.jdo.option.LinkedList	No
javax.jdo.option.TreeMap	No
javax.jdo.option.TreeSet	Yes
javax.jdo.option.Vector	Yes
javax.jdo.option.Map	No
javax.jdo.option.List	No
javax.jdo.option.Array	Yes
javax.jdo.option.NullCollection	No
javax.jdo.option.ChangeApplicationIdentity	No
javax.jdo.option.JDOQL	Yes

PSE Pro Features

There are a number of features included in the native PSE Pro interface that are also supported by the PSEJDO interface. The following table lists the native PSE Pro features and indicates whether or not PSEJDO supports them.

PSE Pro Feature	*PSEJDO Support*
Persistent Arrays	No
Indexes	No
Persistent Garbage Collection	Yes
`osjcheckdb` utility	Yes
`osjshowdb` utility	Limited, no query support
ObjectStore Browser	Yes
Allow Methods in Queries	Yes, except for the `contains()` method on collection types

Contact Progress Software Corporation regarding additional feature support in upcoming releases. As with the optional JDO features we anticipate this list to change substantially in upcoming releases.

Persistent Garbage Collector

The Database Garbage Collector tool (`osjgcdb`) provided with PSE Pro will also work on jdo databases. The tool will reclaim space occupied by deleted objects (via `PersistenceManager.deletePersistent`) in the database, if there are no other objects in the database containing references to these deleted objects. For more information on garbage collecting, see the `osgc` utility in *Managing ObjectStore*.

TreeSets

ObjectStore adds support for TreeSet collections. In OSJDO a `java.util.TreeSet` is converted to a `com.odi.jdo.mutable.Osjdo.TreeSet`. The ObjectStore support for TreeSets is designed for holding large numbers of objects and provides indexing capabilities for better query processing performance.

The OSJDO implementation of TreeSets does not yet provide support for the `SortedSet` interface. This will be fixed in a future release.

Adding Indexes to TreeSets

OSJDO provides an API for adding and managing indexes for `TreeSets`. The following methods are available in the `OsjiHelper` class:

- `void addIndex(Collection c, Class elementType, String path);`
- `void addIndex(Collection c, Class elementType, String path, boolean ordered, boolean duplicates);`
- `void addIndex(Collection c, Class elementType, String path, boolean ordered, boolean duplicates, Placement place);`
- `void addToIndex(Collection c, Object value);`
- `void addToIndex(Collection c, Class type, String path, Object value);`
- `boolean dropIndex(Collection c, Class type, String path);`
- `IndexMap getIndex(Collection c, Class type, String path, boolean ordered);`
- `IndexDescriptorSet getIndexes(Collection c);`
- `IndexMap getSuperIndex(Collection c, Class type, String path, boolean ordered);`
- `boolean hasIndex(Collection c, Class type, String path,  boolean ordered);`
- `void removeFromIndex(Collection c, Class type, String path, Object value);`
- `void removeFromIndex(Collection c, Object value);`
- `void updateIndex(Collection c, Class type, String path, Object oldKey, Object newKey, Object value);`

The behavior of these methods is similar to the corresponding methods in the native PSE Pro interface. For more information, see Enhancing Query Performance with Indexes on page 161.

Example Application

The sample program stores information about a few people, then retrieves some of the information from the database and displays it. The program shows the components you must include in your application so that it can use the PSEJDO interface to PSE Pro. The important steps in creating PSEJDO applications are:

1. Before you begin make sure the environment variable `OS_PSE_ROOTDIR` is set to the PSE Pro installation directory and that the `PATH` environment variable contains `%OS_PSE_ROOTDIR%/bin` on Windows or `$OS_PSE_ROOTDIR/bin` on UNIX.
2. Create a `PersistenceManagerFactory`; this example uses `JDOHelper.getPersistenceManagerFactory()` and uses `jdo.properties` to set various runtime configuration properties.
3. Acquire a `PersistenceManager` by invoking `PersistenceManagerFactory.getPersistenceManager()`. The `PersistenceManager` has the characteristics defined by the properties that are used to create the `PersistenceManagerFactory`.
4. Create a `Transaction` by invoking `PeristenceManager.currentTransaction()`.
5. Start and commit a transaction by calling `Transaction.begin()` and `Transaction.commit()`. The example uses one transaction to populate the database and a second transaction to read and display database data.
6. In the example store the first object by using the `PersistenceManager.makePersistent()` call. Other objects are stored in the database simply because they are reachable from the object that was made persistent.
7. Retrieve and display persistent objects by accessing the `Person Extent`.
8. At the end the example invokes `PersistenceManager.close()`.
9. Run the PSEJDO enhancer on the compiled class files to generate an annotated version of the class files. The annotated version of the class definition is persistence capable.

Example Code

```
package com.odi.demo.jdo.people;

import java.io.InputStream;
import java.io.FileInputStream;
import java.util.Properties;
import java.util.Iterator;
import java.util.Collection;
import java.util.Vector;

import javax.jdo.*;

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

```
  Vector children;

/**
 * Command line: com.odi.demo.jdo.people.Person <create | read>
 */

  public static void main(String argv[]) {

    // process command line arguments
    String operation = null;
    String propFileName =
        System.getProperty("jdo.properties","jdo.properties");

    if (argv.length > 0)
      operation = argv[0];
    if (argv.length > 1 ||
      !("create".equals(operation) ||
      "read".equals(operation))) {
      System.out.println
      ("Usage: people.Person <create | read>");
      System.exit(-1);
    }

    // acquire the PersistenceManagerFactory
    PersistenceManagerFactory pmf = null;
    try {
      InputStream propStream =
        new FileInputStream(propFileName);
      Properties props = new Properties();
      props.load(propStream);
      pmf = JDOHelper.getPersistenceManagerFactory(props);
    } catch(Throwable ex){
      ex.printStackTrace(System.out);
      throw new RuntimeException("Could not create PMF from
        property file");
    }

    // generate a PersistenceManager from the factory
    PersistenceManager pm = pmf.getPersistenceManager();

    // do the requested operation
    if ("create".equals(operation))
      populate(pm);
    else if ("read".equals(operation))
      traverse(pm);
    else
      ;  // not reached

    pm.close();
  }

  static void populate(PersistenceManager pm) {
    Transaction tx = pm.currentTransaction();
    tx.begin();

    // make a network of regular transient Person objects
    Vector kids = new Vector();
    kids.addElement(new Person("Riley", 6, null));
    kids.addElement(new Person("Isabel", 3, null));
    Person guy = new Person("Guy", 47, kids);
```

```
    // now mark the parent object as persistent
    pm.makePersistent(guy);

    // through reachability analysis, upon commit the parent
    // and children and the Vector of children
    // are written into the database.
    tx.commit();
    System.out.println("Created and populated database.");
  }

  static void traverse(PersistenceManager pm){
    Transaction tx = pm.currentTransaction();
    tx.begin();

    // look up the Person object that represents the parent
    // by querying the Extent of the Parent class
    Extent personExtent = pm.getExtent(Person.class, false);
    Query q = pm.newQuery(personExtent, "name == \"Guy\"");
    Collection persons = (Collection)q.execute();

    // assume that there is one and only one object
    // returned from the query
    Person guy = (Person)persons.iterator().next();

    // now access the persistent objects directly
    // as if they were ordinary transient objects
    Vector kids = guy.getChildren();

    System.out.print(guy.getName() + " is " + guy.getAge() +
      " and has " + kids.size() +  " children: ");
    for (int i = 0; i < kids.size(); i++) {
      Person kid = (Person)kids.elementAt(i);
      if (i > 0) System.out.print(", ");
      System.out.print(kid.getName() + " age " + kid.getAge());
    }
    System.out.println("");
    tx.commit();
  }

  public Person(String name, int age, Vector children) {
    this.name = name;
    this.age = age;
    this.children = children;
  }

  public Person() {
    name = null;
    age = 0;
    children = null;
  }

  public String getName() {return name;}
  public int getAge() {return age;}
  public Vector getChildren() { return children; }
}
```

Example Persistence Descriptor

The XML metadata for the example application is contained in persistence descriptor file named `Person.jdo`. In this example, the `Person` class is included in the metadata, so it is `PersistenceCapable`. It has an `identity-type` of `datastore` and its `requires-extent` is set to `true`. The class includes three fields, `name`, `age`, and `children` that are specified as `persistent`.

The example uses the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="com.odi.demo.jdo.people">
    <class name="Person"
        identity-type="datastore" requires-extent="true">
      <field name="name" persistence-modifier="persistent"
          default-fetch-group="true"/>
      <field name="age" persistence-modifier="persistent"
          default-fetch-group="true"/>
      <field name="children" persistence-modifier="persistent"
          default-fetch-group="true"/>
    </class>
  </package>
</jdo>
```

Example Properties File

The example application uses information contained in a file called `jdo.properties` to configure the PSEJDO implementation for the example. It sets up the class to be used for `PersistenceManagerFactory`. It also uses ObjectStore specific extensions to configure the application to use the PSE Pro product. The file contains the following:

```
javax.jdo.PersistenceManagerFactoryClass=
      com.odi.jdo.OsjdoPersistenceManagerFactory
javax.jdo.option.ConnectionURL=
      osjdo:C:\ODI\PSEProJ\com\odi\demo\jdo\people\person.odb
com.odi.jdo.option.ConnectionOpenMode=update
com.odi.jdo.option.Product=PSEPro
com.odi.jdo.option.ConnectionOpenAction=auto
```

Before You Run the Program

Before you can run the sample program, you need to:

- Add additional files to the `CLASSPATH` environment variable
- Compile the source file
- Run the enhancer on the `.class` file

Adding Additional Files to the CLASSPATH

In your `CLASSPATH` environment variable, specify the required jar files as described in Adding Files to Your CLASSPATH on page 246.

Compiling the Program

To compile the example program, change to the PSE Pro installation directory and, on Windows platforms, enter

```
javac com\odi\demo\jdo\people\*.java
```

On UNIX platforms, enter

```
javac com/odi/demo/jdo/people/*.java
```

As output, the `javac` compiler produces the byte code class file `Person.class` in the same directory that contains the `Person.java` file

Running the Enhancer

You need to run the JDO enhancer to modify the `Person` class so that it implements the `PersistenceCapable` interface and annotate the `Person` class methods to manage the lifecycle of a `peoplePerson` interface in the JVM. The JDO enhancer generates new annotated class files. After you run the postprocessor, you should use the annotated class files rather than the original class files.

The following command can be used to run the JDO enhancer on the example application on Windows:

```
osenhance -dest C:\ODI\PSEProJ\classes
    com.odi.demo.jdo.people.Person
```

On UNIX, use the following:

```
osenhance -dest /ODI/PSEProJ/classes
    com.odi.demo.jdo.people.Person
```

Using the `-dest` option specifies that you want to place the newly annotated class files in a `classes` directory under the PSE Pro installation.

The final argument, `com.odi.demo.jdo.people.Person`, specifies the name of the class in the example you need to be make `PersistenceCapable`.

This command will modify the `Person.class` file as it finds it in the `java` classpath.

Running the Program

Run the example program as a Java application. On Windows platforms, run it with the following command:

```
java -classpath C:\ODI\PSEProJ\classes;%CLASSPATH%
  -Djdo.properties=
  C:\ODI\PSEProj\com\odi\demo\jdo\people\jdo.properties
  com.odi.demo.jdo.people.Person create
```

Run the example program on UNIX with the following command:

```
java -classpath /opt/ODI/PSEProJ/classes:${CLASSPATH}
 -Djdo.properties=
 /opt/ODI/PSEProJ/com/odi/demo/jdo/people/jdo.properties
 com.odi.demo.jdo.people.Person create
```

This will create three new files that represent the PSE Pro database, `person.odb, person.odf, person.odt`. See Chapter 4, Managing Databases, on page 59 for an explanation of these files. For simplicity the user should make sure these files are treated as a cohesive unit for backup and relocation purposes.

To see the results of the creation, on Windows platforms execute:

```
java -classpath C:\ODI\PSEProJ\classes;%CLASSPATH%
  -Djdo.properties=
  C:\ODI\PSEProj\com\odi\demo\jdo\people\jdo.properties
  com.odi.demo.jdo.people.Person read
```

On UNIX platforms execute:

```
java -classpath /opt/ODI/PSEProJ/classes:${CLASSPATH}
 -Djdo.properties=
 /opt/ODI/PSEProJ/com/odi/demo/jdo/people/jdo.properties
 com.odi.demo.jdo.people.Person read
```

The expected output is

```
Guy is 47 and has 2 children named: Riley age 6, Isabel age 3
```

Chapter 12
Miscellaneous Information

This chapter provides miscellaneous information about PSE Pro.

Contents

This chapter discusses the following topics:

Java-Supplied Persistence-Capable Classes

Some Java-supplied classes are persistence capable. Others are not persistence capable and cannot be made persistence capable. A third category of classes can be made persistence capable, but there are important issues to consider when you do so.

Description of Java-Supplied Persistence-Capable Classes

The following Java classes are persistence capable:

- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.util.Locale`
- `java.util.Currency`

The wrapper classes follow:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`

- `java.lang.Short`
- Arrays of `Object`, of any of the primitive types (`boolean`, `byte`, `integer`, and so on), and of any persistence-capable type are all persistence capable. (You can allocate an array and initialize it later, just as you would with any other field.)

Identity

PSE Pro does not always preserve identity for objects that are instances of the Java wrapper or `String` classes. It is more efficient to store these objects as values rather than as objects. Because identity is not always preserved, programs that use object identity to compare wrapper class objects work differently when used with persistent objects. For example, the following method is incorrect:

```
boolean comparePersistIntegers(Integer x, Integer y) {
  return (x == y);
}
```

Instead, it should be written as

```
boolean comparePersistIntegers(Integer x, Integer y) {
  return x.equals(y);
}
```

Additional information about object identity is in About Object Identity on page 107.

Virtual memory overhead

When PSE Pro makes them persistent, `String` types, primitive wrapper types, and arrays have more run-time virtual memory overhead than types that implement `IPersistent`. This is because PSE Pro must create entries for these types in two hash tables. `com.odi.IPersistent` requires an entry in a single hash table because certain information is stored in fields in the object.

Persistent and persistence capable

In your program, some wrapper objects or strings might be persistent and some might be transient, though both are persistence capable:

PSE Pro always makes objects of type `java.lang.Double` or `java.lang.Long` persistent when it migrates them into a database. This can happen with an explicit call to `ObjectStore.migrate()` or through transitive persistence. All other wrapper objects behave like this:

- If the application explicitly calls `ObjectStore.migrate()` on a wrapper object, or as a value in an `com.odi.util.OSTreeMap`, the wrapper object or string becomes persistent.
- If the wrapper object or string is only reachable through transitive persistence, it does not become persistent when the transaction is committed. Instead, PSE Pro stores the object as an immediate value.

  This means that PSE Pro does not store the object in any of its internal hash tables and does not store the object as a separate value in the database. Instead, PSE Pro stores the object in the location of the reference to the object. The reference completely describes the object.

- Some strings become immediate if you set the `com.odi.useImmediateStrings` property to `true` when a session is created and the string is eight characters or less. These strings are stored as values of fields in persistent objects and are not persistent unless you explicitly call `ObjectStore.migrate` to make them persistent.

Any routine that requires a persistent object, as opposed to a persistence-capable object, notices the distinction between persistent and *persistence capable but transient*. For example, if an application calls `Segment.of()` on an `Integer` object, the return value might be a segment in a database or PSE Pro might signal `ObjectNotPersistentException`. You cannot always predict what the return value will be because an `Integer`-valued field in a persistent object can contain either a persistent or a transient value.

Unicode strings

PSE Pro stores Unicode strings. You can specify any Java string with Unicode characters in it and PSE Pro can store it persistently and retrieve it correctly. PSE Pro uses UTF-8 (Unicode Transformation Format) encoding/compression to store regular English strings compactly. Sun's Java implementation uses the same mechanism.

Immediate strings

The identity of immediate strings is not maintained.

Can Other Java-Supplied Classes Be Persistence Capable?

Many Java system classes cannot be persistence capable. There are other Java system classes that you can make persistence capable but you must consider some issues when you do so. In some situations, you can subclass the Java system class and make the subclass persistence capable. Of course, this would not work for final classes.

Primitive types

You cannot store an object of a primitive type, such as an `int`, directly in a database as a discrete object. To store an object of a primitive type in a database you can

- Place it in a wrapper object, such as an `Integer`
- Define it as a field in a persistence-capable class

For example, you cannot make `byte` persistence capable because by itself a `byte` is not an `Object`, but you can make `byte[]` persistence capable because it is an `Object`.

Native methods

Classes that use native methods cannot be made persistence capable by the postprocessor because the postprocessor cannot annotate the native methods the way it can annotate Java code. What this means is that if a class has native methods and you postprocess the class, PSE Pro cannot guarantee that everything will work properly.

You might choose to postprocess your code, then add native code. If you do this, you must ensure that any persistent objects that your native code references are properly fetched from the database before the native method is called.

Classes that hold state

Other system classes do not make sense as persistent objects because they hold state that is inherently tied to the process, such as open file channels or Java threads.

Post-processing

For other classes, such as `java.lang.Stringbuffer`, the previous obstacles might not apply. If you postprocess the `.class` file for `java.lang.StringBuffer` and specify the `-modifyjava` option, the postprocessor produces a persistence-capable `StringBuffer` class:

```
osjcfp -dest \osjcfpout -modifyjava java.lang.StringBuffer
```

Then you must put the new `.class` file in your `CLASSPATH` variable ahead of the standard Java `.class` file. All subsequent use of the `StringBuffer` class in this environment would use the persistence-capable version.

Performance drawbacks

There are, however, some drawbacks to doing this. Some slowdown of some or all the methods will occur, because the postprocessor must add new instructions to check whether the object needs to be brought in from the database or needs to be marked as modified.

The amount of slowdown is hard to determine. It depends on the details of the method. Even parts of your program that never handle persistence are affected by these extra instructions. This also applies to indirect uses of the class, for example, if `StringBuffer` is used heavily in a Java library that you are using, such as a user interface or network library.

Library version problems

There can also be problems with Java library version skew. If you postprocess `java.lang.StringBuffer` from version 1.1 of the Java Virtual Machine, then your user uses your program with version 1.1.2, and `StringBuffer` has changed in some way between 1.1 and 1.1.2, your user will see the 1.1 version (persistence capable) everywhere in the entire Java environment. If your user was depending, directly or indirectly, on the new 1.1.2 version of `StringBuffer`, something might not work properly.

Renaming the class

You might need to rename the newly created persistence-capable version so that the non-persistence-capable version is still available to the other Java system classes. To do this, specify the -translatepackage option when you run the postprocessor. See Putting Processed Classes in a New Package on page 193.

This avoids the problem and is generally safer. However, you might need the persistence-capable class to have the original class name. For example, suppose you have a library with a method that takes an argument of type `java.lang.Stringbuffer`. You want to pass in a persistence-capable object. You cannot rename the class because the argument type would not match.

java.util.Hashtable

PSE Pro itself uses `java.util.Hashtable`. Consequently, invoking Java or using PSE Pro with a persistence-capable version of `java.util.Hashtable` that is available in your `CLASSPATH` is likely to cause trouble, such as infinite loops. A better approach is to substitute the PSE Pro-supplied class `com.odi.util.OSHashtable`.

Description of Special Behavior of String Literals

There are special considerations when you make `String` literals persistent.

When a Java program refers to a `String` literal by using quotation marks to name a string, Java treats the resulting `String` as a constant value. Multiple calls to a method with the `String` literal operate on the same `String` object.

The com.odi.stringPoolSize initialization property allows you to control the way that PSE Pro causes `String`s, other than literals with the same contents, to be represented by a single, shared instance in the database in certain circumstances. See Description of com.odi.stringPoolSize on page 54.

Example of String Behavior

Consider the following example:

```
Object string() {String result = "string"; return result;}
Object intArray() {int[] result = { 1, 2, 3 }; return result;}
boolean stringsTheSame() {return string() == string();}
boolean intArraysTheSame() {return intArray() == intArray();}
```

The `stringsTheSame()` method always returns `true` because every call to `string()` returns the same `String` object. The `intArraysTheSame()` method always returns `false` because each call to `intArray()` constructs a new `int[]` object.

Destroying Strings

By default, `String` objects that become persistent during a transaction revert to being transient at the end of the transaction. Persistent objects usually are made stale at the end of a transaction. Unlike objects that implement `IPersistent`, when a `String` is made stale, it becomes transient.

When you destroy a `String`, in the transaction in which the destroy operation occurs, PSE Pro keeps track of the fact that the object was destroyed. An attempt to use a destroyed `String` literal causes PSE Pro to signal `ObjectNotFoundException`. The solution is to copy the `String` before you destroy it.

You should not destroy a `String` in a database unless you know that no other object in the database refers to that `String`. A safe, though possibly inefficient, way to handle this is to use

```
new String(String)
```

to force a new identity to each `String` that might be referenced. Also, you must disable the `String` pool by specifying `0` for the value of the `com.odi.stringPoolSize` initialization property. This ensures that you can safely destroy the old `String` instance.

It is usually best to avoid destroying strings or objects altogether and let the persistent garbage collector take care of destroying such unreachable objects. The persistent GC can typically destroy and reclaim such objects very efficiently, because it can batch such operations and cluster them effectively. If you set up the GC to run when the system is lightly loaded, you can defer the overhead of the destroy operations to a time when your system would otherwise be idle, thus getting greater real throughput from your application when you really need it.

Immediate strings

You do not need to destroy immediate strings. Destroying an immediate string has no effect. Immediate strings are not persistent objects. They are transient values that are not stored in the database.

Serializing Persistent Objects

You can serialize many classes that implement `com.odi.IPersistent`. For this to work, the definition of your persistence-capable class must implement the `java.io.Serializable` interface. The classes you can serialize include

- `com.odi.util.OSHashtable`
- `com.odi.util.OSTreeMap`
- `com.odi.util.OSVector`
- `com.odi.util.OSTreeSet`

During serialization, none of the transient fields in the `IPersistent` implementation needs to be written out. When you deserialize instances of the `com.odi.util.OSTreeMap` and `com.odi.util.OSTreeSet` classes, you can specify their placement by using the `ObjectStore.setPlacementForSerialization()` method.

Before serializing an object, an application must always invoke `ObjectStore.deepFetch()` on the object to be serialized. The `deepFetch()` method ensures that the contents of all components of the object are accessible. This must be the case for an application to serialize an object.

Why you use deepFetch()

In a PSE Pro application, the first time you read or modify an object, PSE Pro makes the contents of the object available. The contents do not have to be available before you start the operation. You need not add Java code to make the contents available. When a PSE Pro program follows a reference from a source object to a target object, the contents of the target object are automatically available. This happens because the postprocessor recognizes the Java byte-code instructions that follow references and it inserts the code that fetches the object contents.

Serialization works differently. It follows references from one Java object to another without using Java byte codes. Serialization does not perform the automatic fetches the way that PSE Pro does. Consequently, before you initiate serialization of an object, its contents and the contents of all its components must already be available. The `ObjectStore.deepFetch()` operation does this for you.

Example

When an application serializes and deserializes a persistent object with the default serialization methods, PSE Pro effectively creates a transient copy of the object and its components. Following is code that provides an example of serializing and deserializing persistent objects. In this example, `list2` is a transient copy of the persistent list.

```
public
class SerializationExample {

public static void main(String argv[])
  throws java.lang.ClassNotFoundException, java.io.IOException,
  java.io.FileNotFoundException {
  String dbName = argv[0];
  Session.createGlobal(null, null);
  /* Create a database with a list in it. */
  Database db = Database.create(dbname,
    ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
```

```
  Transaction tr = Transaction.begin(ObjectStore.UPDATE);
  List curr = new List("1", null);
  db.createRoot("list", curr);
  for (int i=2; i < 5; i++) {
    curr.next = new List(""+i, null);
    curr = curr.next;
   }
  tr.commit();

  /* Illustrate use of serialization in this example. */
  tr = Transaction.begin(ObjectStore.UPDATE);
  List head = (List)db.getRoot("list");

  /* Fetch the entire list prior to serializing it. */
  ObjectStore.deepFetch(head);

  FileOutputStream f = new FileOutputStream("tmp");
  ObjectOutputStream os = new ObjectOutputStream(f);
  os.writeObject(head);

  FileInputStream in = new FileInputStream("tmp");
  ObjectInputStream is = new ObjectInputStream(in);

  /* list2 is effectively a copy of the list denoted by head. */
  List list2 = (List)is.readObject();
  ...

  tr.commit();
  }
}

public class List implements java.io.Serializable {

  public Object value;
  public List next;

  List(Object value, List next) {
    this.value = value;
    this.next = next;
  }

  ...
}
```

Using Persistence-Capable Classes in a Transient Manner

The `stublib.jar` file contains stubs of PSE Pro classes that allow user-defined persistence-capable classes to be used in a purely transient manner. The annotations in the persistence-capable classes make calls to the various PSE Pro stub routines in `stublib.jar`.

The `stublib.jar` file provides a stripped-down version of the PSE Pro API. This allows better performance and a smaller footprint than the complete `.jar` file.

For example, you might want to use `stublib.jar` for the client in an RMI or CORBA application. The client might use persistence-capable classes, which make references to various PSE Pro methods, but the client never directly accesses a PSE Pro database. In this situation, the stub routines in `stublib.jar` satisfy the requirements of the Java VM's linker.

To use `stublib.jar`, put it in your `CLASSPATH` instead of `pro.jar`.

If your application uses any classes in `com.odi.util`, you must use `pro.jar`. You cannot use `stublib.jar` because the stub definitions are not sufficient for the `com.odi.util` classes.

Java primitive fields

Java primitive fields are represented the same way as similarly sized C++ primitives. The sizes of Java primitives when they are stored in instance fields or arrays are shown in the following table.

Primitive	*Primitive Size in Bytes*
`boolean`	1
`byte`	1
`short`	2
`char`	2
`int`	4
`float`	4
`double`	8
`long`	8

Environment Variables

PSE Pro includes the following environment variables:

- `OS_JAVA_VM` specifies the command for running the Java virtual machine, when it is set. The default is that this variable is not set. The Windows tool batch files use the value of this variable when it is set.
- `OSJCFPJAVA` specifies the name of the Java executable you want the postprocessor to use. The default is `java`. If this variable is not set, the postprocessor uses the first Java executable that it finds in your `PATH` environment variable. If you want the postprocessor to use another Java executable, set the `OSJCFPJAVA` environment variable to the name of the Java executable you want the postprocessor to use.

  If the postprocessor cannot find a Java executable, it signals a `Bad command or file name` error message.

Chapter 13
Tools Reference

This chapter provides reference information for the following tools:

osjcfp: Running the Postprocessor

To make classes persistence capable, compile the source files, then run the postprocessor on the resulting class files. You must run the postprocessor on all class files in a batch at the same time. The postprocessor can accept a command line that intersperses file names, options, and input file specifications. Complete information about the postprocessor is in Chapter 8, Generating Persistence-Capable Classes Automatically, on page 171.

Command Format

```
osjcfp -dest destination_dir file_name [file_name...][options].
```

Options

`@input_file`

Causes the contents of the named input file to replace this argument in the command line. The postprocessor does this before any other argument processing. You can specify this option multiple times on one command line to include multiple files. You cannot nest this option. That is, the postprocessor does not expand this argument if it appears in an input file.

`-annotatefield qualified_field_name`

Instructs the postprocessor to insert `fetch()` or `dirty()` calls in annotated code whenever a method accesses a transient field in a persistent class. The `-annotatefield` option turns off the `-noannotatetransientfields` option.

`{ -cis | -classinfosuffix } suffix_string`

Specifies the suffix that the postprocessor adds to the name of the `ClassInfo` subclass that the postprocessor generates for each class that it makes persistence capable. By default, the suffix is `ClassInfo`. This is useful when you need to limit the number of characters in file names. For all batches in an application, you must specify the same suffix if you do not use the default.

`{ -cpath | -classpath } class_path`

Specifies the path by which to locate class files for postprocessing. If you specify this option, PSE Pro uses it in place of the `CLASSPATH` environment variable. The `-classpath` option does not affect the class path used to run the postprocessor.

`{ -sysclasspath | -syscpath} system_class_path`

Specifies the path by which to locate class files for Java system classes for postprocessing. If you specify this option, PSE Pro uses it in place of the value in the `sun.boot.class.path` system property. When using `jview`, system classes cannot be found unless you specify their location using this option. The `-sysclasspath` option does not affect the class path used to run the postprocessor.

`{ -cc | -copyclass }`

Copies classes to the destination directory without annotating them. This option applies to class names, `.class` files, `.jar` files, and `.zip` files that you specify on the command line after the `-copyclass` option and before the next `-persistcapable` or `-persistaware` option or the end of the command line. This option is useful when you want nonpersistent classes or classes that have already been annotated to be in the same directory as persistence-capable or persistence-aware classes being created.

If you specify the `-persistaware` or `-persistcapable` option for any file for which you also specify the `-copyclass` option, the postprocessor ignores the `-copyclass` option for that file.

If you specify the `-translatepackage` option and the `copyclass` option, the postprocessor modifies the class to accommodate the new package name.

`{ -d | -dest }` *`destination_dir`*

This option is required. The postprocessor uses the directory you specify for *`destination_dir`* as the root for locating the annotated files. The postprocessor places each class file it operates on in the package-appropriate subdirectory of the destination directory, as though the destination directory were in your class path.

If the destination directory specification would cause the postprocessor to overwrite an original file and you did not specify the `-inplace` option, the postprocessor reports an error and terminates without producing any output.

`{ -emlf | -embeddedmaxlengthfield }`

Specifies that Java Strings for a given field will be represent as embedded Unicode arrays. Using embedded strings can significantly improve performance for those applications that heavily use small strings (less than 100 characters).

This option takes the following two arguments:

- First argument is the fully qualified name of a string field
- Second argument is the maximum size string that can be stored in that field.

Note that you cannot declare embedded string fields as indexable fields for Java peer collections.

`{ -f | -force }`

Forces the postprocessor to overwrite existing annotated `.class` and `ClassInfo` files.

`-hashcode` *`class_name`*

Causes the postprocessor to add a persistent `hashCode()` method to the specified class. You typically use this option with the `-nodefaulthashcode` option. If you specify this option for a class for which you explicitly defined a `hashCode()` method, the postprocessor reports an error.

`-includesummary` *`inc_class_name`*

Instructs the postprocessor to include the specified summary in the new summary it creates. It does not matter whether the postprocessor is also annotating any classes. Replace *`inc_class_name`* with the name of a file generated by a previous execution of the postprocessor with the `-summary` option. You must know the name of the generated file. If you did not postprocess the library yourself, you must get the name of the generated class from the library vendor. When you specify the `-includesummary` option, you must also specify the `-summary` option. This option provides information that is needed by the schema evolution API, which is not available in PSE Pro. You can, however, use serialization to perform schema evolution.

`{ -index | -indexablefield }` *`field`*

Marks a field as indexable for a peer (`com.odi.coll`) collection. This option applies only to the field that immediately follows it. You must specify a fully qualified field, for example, `com.odi.demo.people.name`. You can specify this option multiple times. This option does not apply to utility (`com.odi.util`) collections.

This option is useful because it allows a query to run faster. The postprocessor does not actually add the index. You can add a persistent index with a call to the API at run time. When the index is present, queries that use the specified field are faster.

Suppose you declare a field to be indexable, then you change the value of that field. Performance is slightly slower than if the field were not indexable. This is true for any object of the class, whether or not the object is in a collection.

Now suppose an object with an indexable field is in a collection and the collection has an index on the indexable field. If you change the value of the field, performance is slightly slower than when the object is not in a collection. The extra time is needed to update the index.

An object can belong to many collections. Each collection can have an index on a particular field. If you change the value of that field, ObjectStore must update each index and the performance penalty is greater.

If a class has any indexable fields, every instance of the class is larger by three 32-bit words in the database.

`-inplace`

Causes the postprocessor to annotate stand-alone files — files that are not in `.zip` files or `.jar` files — in place rather than writing the annotated file in the destination directory. When the postprocessor annotates a class in place, it overwrites the original class files with the annotated class files and writes the `ClassInfo` subclass to the same directory as the persistence-capable class. If a class originates in a `.zip` file or `.jar` file, the postprocessor writes the annotated class and its corresponding `ClassInfo` subclass to the destination directory.

Do not use this option when you are doing iterative development. During development, a separate output directory avoids errors and supports debugging.

When you specify the `-inplace` option, you must still specify a destination directory, but the postprocessor ignores it for stand-alone files.

`{ -it | -ignoretransient }` *`field_name`*

Instructs the postprocessor to ignore the transient attribute of the specified field and treat the field as a persistence-capable field. You must specify a fully qualified field name. The field is treated as persistence capable only for purposes of postprocessing. This option is useful when a persistence-capable class you are defining inherits from a class that includes a transient field. If you do not specify this option for a transient field, the postprocessor ignores the field, which can cause problems if you want to use the field.

`-modifyjava`

Allows the postprocessor to modify classes in standard Java packages. The default is that the postprocessor does not modify standard Java classes.

`{ -naf | -noannotatefield }` *`qualified_field_name`*

Prevents access to the specified field from causing `fetch()` and `dirty()` calls on the containing object. This is useful for transient fields when you access them outside a transaction. Normally, access to a transient field causes `fetch()` or `dirty()` to be called to allow the `postInitializeContents()` and `preFlushContents()` methods to convert between persistent and transient state.

`{ -natf | -noannotatetransientfields}`

Prevents the postprocessor from inserting `fetch()` or `dirty()` calls into the annotated code for transient fields of persistent classes that are accessed by methods. This option is useful when you want to access the transient fields outside of transactions.

The default behavior, if the`noannotatetransientfields` option is not specified, is for the postprocessor to insert `fetch()` or `dirty()` calls into the annotated code when transient fields of persistent classes are accessed.

`-noarrayopt`

Disables optimization of `fetch()` and `dirty()` calls for array objects in looping constructs. This causes `osjcfp` to insert the calls to `fetch()` or `dirty()` in every iteration rather than only in the first loop iteration.

`{-nodefaulthashcode | -ndhc }`

Prevents the postprocessor from automatically adding a `hashCode()` method to a class, except classes for which you explicitly specify the `-hashcode` option. If you specify this option, it is your responsibility to ensure that there is a suitable `hashCode()` method for classes that are used as keys in persistent hash tables.

`-noinitializeropt`

Disables optimization of `fetch()` and `dirty()` calls in constructors. Specify this option when you want the postprocessor to perform full annotation of constructors. Full annotation means that if the object becomes persistent during constructor execution, modifications to the object are handled correctly. By

default, the postprocessor does not fully annotate constructors to handle changes in the newly constructed object. Typically, this is the desired behavior.

If your application inserts objects into PSE Pro collections during construction of the objects being inserted, you must specify the `-noinitializeropt` option. Doing so avoids errors in the handling of modifications to the newly constructed objects.

`-noopt`

Disables the three optimizations that are disabled by the `-noarrayopt`, `-noinitializeropt`, and `-nothisopt` options. The `-noopt` option is a shortcut you can use when you want to specify all three options. You might want to specify this option when the optimizations are preventing the postprocessor from inserting required `fetch()` and `dirty()` calls in your classes.

`-nooptimizeclassinfo`

Instructs the postprocessor to generate *xxx*`ClassInfo` classes for persistence-capable classes that are public. Use this option if your application encounters run-time security errors when the *xxx*`ClassInfo` classes are dynamically generated using the reflection API.

`-nothisopt`

Disables optimization of `fetch()` and `dirty()` calls for access to fields relative to `this` in nonstatic member methods. This causes `osjcfp` to insert a `fetch()` or `dirty()` call for each access to a field in `this`.

`-nowrite`

Performs process and error checking but does not actually annotate class files. This option allows a test run of the postprocessor. You use it to determine whether or not all specified classes are accessible, whether additional options are needed, and if you specify `-v` (verbose), you can see where the resulting files would be located.

`{ -pa | -persistaware }`

Causes subsequent `.class` files, `.jar` files, and `.zip` files on the command line to be persistence aware. This means that instances of the classes can operate on persistent objects but cannot be persistent. The postprocessor annotates persistence-aware classes so that there are calls to `ObjectStore.fetch()` and `ObjectStore.dirty()` where needed during operations on potentially persistent objects and arrays that might be used by the persistence-aware class. This option applies to class names, `.class` files, `.jar` files, and `.zip` files that you specify on the command line after the `-persistaware` option and before the next `-persistcapable` or `-copyclass` option, or the end of the command line. In other words, the `-persistcapable` option or the `-copyclass` option alters this mode. The `-pc` option is in effect by default.

`{ -pc | -persistcapable }`

Causes subsequent `.class` files, `.jar` files, and `.zip` files on the command line to be persistence capable. This option applies to class names, `.class` files, `.jar` files, and `.zip` files that you specify on the command line after the

`-persistcapable` option and before the next `-persistaware` or `-copyclass` option or the end of the command line. The `-pa` (persistence-aware) option or the `-copyclass` option alters this mode. The `-pc` option is in effect by default.

`{ -q | -quiet }`

Causes the postprocessor to refrain from displaying warnings. A warning message provides information about something that the postprocessor recognizes as a possible problem but cannot confirm as actually being a problem. This option cancels a previous `-verbose` option, if you specified one.

`{ -qc | -quietclass }` *`class_name`*

Causes the postprocessor to refrain from displaying warnings for the specified class. A warning message provides information about something that the postprocessor recognizes as a possible problem but cannot confirm as actually being a problem. This option applies only to the name that immediately follows it. Specify a fully qualified class name. If the postprocessor is renaming the class, it does not matter whether you specify the old name or the new name. If you specify `-verbose` in the same command, this option takes precedence over the specified class.

`{ -qf | -quietfield }` *`member_name`*

Causes the postprocessor to refrain from displaying warnings for the specified class field. A warning message provides information about something that the postprocessor recognizes as a possible problem but cannot confirm as actually being a problem. This option applies only to the name that immediately follows it. Specify a fully qualified class field name. If the postprocessor is renaming the class, it does not matter whether you specify the old name or the new name. If you specify `-verbose` in the same command, this option takes precedence for the specified class field.

`-summary` *`gen_class_name`*

Causes the postprocessor to generate a class with the name *`gen_class_name`*. This generated class extends the `com.odi.PersistentTypeSummary` class. The *`gen_class_name`*`.getPersistentClasses()` method returns a list of the classes that were made persistence capable in this execution of the postprocessor. The generated class has a no-argument constructor that passes arrays to the `PersistentTypeSummary` class constructor. To identify persistence-capable classes in libraries and include them in this summary, specify the `-includesummary` option. This option provides information that is needed by the schema evolution API, which is not available in PSE Pro. You can, however, use serialization to perform schema evolution.

`{ -tf | -transientfield }` *`qualified_field_name`*

Causes the postprocessor to treat the specified field as though it has a `transient` modifier, even if it does not. This typically is useful when a field should not be stored in a database, but it must be available for object serialization.

`{ -tp | -translatepackage }` *`orig_pkg_name new_pkg_name`*

Renames classes that belong to *`orig_pkg_name`* so that they belong to *`new_pkg_name`*. The original `.class` files remain in the original location and the postprocessor does not annotate them. For example, suppose the postprocessor makes a class named `a.b.C` persistent with `-tp a.b a.b.x`. The persistent class has the name `a.b.x.C`.

A package specification of "`.`" implies the default unnamed package. For example, the option `-tp . persist` causes the unpackaged class name `C` to be renamed `persist.C`.

`orig_pkg_name` must exactly match the package name of the class being annotated. For example, for a file named `a.c.D`, a specification of `-tp a a.b` does not translate the package name. The package of `a.c.D` is `a.c`, not `a`.

The postprocessor changes the package name of all classes in the original package that it can locate through the `CLASSPATH` environment variable or, if it is specified, the `-classpath` option.

`{ -v | -verbose }`

Causes the postprocessor to write descriptions of its actions to standard output. This option cancels a previous `-quiet` option, if you specified one.

File Names

You can specify any number of class names, `.class` files, `.jar` files, or `.zip` files on the command line. The postprocessor recognizes files as follows:

`name``.class`

Explicit file name for a class file.

`name``.zip`

Explicit file name for a class `.zip` file. The postprocessor processes all `.class` files in the `.zip` file according to the persistence mode that is in effect when the postprocessor encounters the name of the `.zip` file. The postprocessor places each file from the `.zip` file in the package-relative subdirectory of the destination directory. A `.zip` file allows the postprocessor to process multiple files without the specification of each one on the command line. Also, you can simply specify a `.zip` file. You need not unzip the file before processing the `.class` files.

`name``.jar`

Explicit file name for a class `.jar` file. The postprocessor treats the file the same way that it treats a `.zip` file.

`name`

Qualified class name delimited by "`.`". The postprocessor uses the `CLASSPATH` environment variable or the specification for the `-classpath` option to locate the `.class` file, which can be in a `.zip` file or a `.jar` file.

Because the postprocessor recognizes *`name`*`.class` as well as *`name`*, you can run a command such as

```
osjcfp -dest osjcfpout *.class
```

You need not derive qualified class names from the file paths.

Postprocessor API

PSE Pro also includes a companion API for it class file postprocessor utility for those times when it is useful to call the postprocessor from your Java application. The method that you call is

`com.odi.filter.OSCFP.filter(String[ ] argv)`

The arguments you pass with the *argv* parameter are the same as the arguments you would use for the `osjcfp` command-line utility. These arguments are listed in the preceding sections. The following example shows how to use the postprocessor API:

```
String args = new String[numArgs];
args[0] = "-inplace"
args[1] = "-dest";
args[2] = ".";
 ...;
filter.OSCFP myPostprocessor = new filter.OSCFP();
myPostprocessor.filter(args);
```

Note

When using this method, you cannot use wildcards with file names. For example, on the command line, `*.class` expands to all the `.class` files listed in your current directory; however, it does *not* expand when used with the API. You must specify each file name explicitly.

osjcheckdb: Checking References in a Database

The `osjcheckdb` utility or the `Database.check()` method checks the references in a database. This tool scans a database and checks that there are no references to destroyed objects. You can fix references to destroyed objects by finding the objects that contain the dangling references and overwriting the invalid references with something else, such as a null value. In addition to finding references to destroyed objects, the tool performs various consistency checks on the database.

If the tool does not find any problems, it does not produce any output.

Check paths

Before you invoke `osjcheckdb` from the command line, ensure that `tools.jar` is in your `CLASSPATH` variable. Also ensure that the distribution `bin` directory that contains `osjcheckdb` is in your `PATH` variable. The format for invoking this tool from the command line is

Command line

`osjcheckdb [-openUpdateForRecovery]`*`database_name1`*`.odb ...`

Optionally, specify the `-openUpdateForRecovery` flag. If you specify this option and a database to be checked needs to be recovered before it can be checked, PSE Pro recovers the database and then checks references. If you do not specify this option, and a database to be checked needs to be recovered, PSE Pro throws `com.odi.DatabaseNeedsRecoveryException`. If no database to be checked needs to be recovered, it does not matter whether or not you specify this option.

You can specify one or more databases. Separate multiple specifications with a space. If `osjcheckdb` cannot check a database that you specify, it displays a message about the inaccessible database and continues to the next database.

Be sure to specify the name of the `.odb` file of the database.

The tool displays messages about any errors that it finds.

API

The function signature for invoking the API for this tool is

```
Database.check(java.io.PrintStream)
```

When PSE Pro executes this method, it operates on the committed contents of the database and on any changes that have been saved as a result of `ObjectStore.evict()` or `ObjectStore.evictAll()`. PSE Pro does not operate on any changes that have been made but not committed or evicted.

The method writes any errors it finds to the argument stream. It also returns a Boolean value, which is `true` if the references are valid and `false` if there are any bad references.

osjgcdb: Collecting Garbage in Databases

The command-line utility for collecting garbage in a database is `osjgcdb`. Invoke this tool with the following format:

`osjgcdb` [`-detail`] *`database_name`*`.odb ...`

You can specify multiple database names. When you specify the `-detail` option, PSE Pro displays information about the properties used to perform the garbage collection. For example, execution of

```
osjgcdb db.odb
```

is the same as calling the `Database.GC()` method on the `db.odb` database. Here are two examples:

```
osjgcdb \temp\mumble.odb
Reclaimed 1 unreachable objects.
```

```
osjgcdb -detail \temp\mumble.odb
Reclaimed 1 unreachable objects.
-- listing properties --
Rescan Passes=1
Unreferenced Dead Objects=1
Unreferenced Objects=1
Rescan Objects=0
Unreferenced Live Objects=0
com.odi.gc.reclaimedObjects=1
```

The `osjgcdb` tool requires

- The `tools.jar` file in your `CLASSPATH` environment variable
- The PSE Pro `bin` directory in your `PATH` environment variable

Not PSE

This tool and its API are not provided with PSE.

osjshowdb: Displaying Information About a Database

The `osjshowdb` utility displays information about one or more databases. This utility is useful when you want to know how many and what types of objects are in a database. You can use this utility to verify the general contents of the database.

This utility displays the following information:

- Name of the database
- Name and number of each type of object in the database
- Total size in bytes occupied on the disk by each type of object
- Number of destroyed objects

-showObjs option

If you specify the `-showObjs` option, the `osjshowdb` utility also displays the following information for each object:

- `oid`, which is an internal representation of its location in the database
- Type
- Number of bytes it occupies on the disk
- If it is an array, the number of elements in the array

-showData option

Specify the `-showData` option with `osjshowdb` to display string values as well as the references that an object contains. When you specify the `-showData` option, it implies the `-showObjs` option.

Path variables

Before you invoke `osjshowdb` from the command line, ensure that `tools.jar` is in your `CLASSPATH` environment variable. Also ensure that the distribution `bin` directory that contains `osjshowdb` is in your `PATH` variable.

Command line

To execute the `osjshowdb` utility, use this format:

```
osjshowdb [-showData] [-showObjs] db1.odb [db2.odb]...
```

You can specify one or more databases.

When the utility displays `java.lang.String` objects, the number of elements is the number of characters in the string. The total bytes indicates the number of bytes that the data occupies on the disk.

There are internal structures in the database that are not included in the calculations performed by the `osjshowdb` utility. Consequently, the total number of bytes as indicated in the output from `osjshowdb` is never equal to the actual size of a segment.

API

The API for the `osjshowdb` utility is `Database.show()`.

When PSE Pro executes this method, it operates on the committed contents of the database and on any changes that have been saved as a result of `ObjectStore.evict()` or `ObjectStore.evictAll()`. PSE Pro does not operate on any changes that have been made but not committed or evicted.

osjup70: Upgrading Databases to 7.0 Format

Release 7.0 changes the format of PSE Pro databases. Consequently, you must upgrade 6.*x* databases before you can use them with 7.0. To upgrade a database, run the `osjup70` utility on that database. This utility is in the `bin` directory of your PSE Pro installation directory. The format for running the `osjup70` utility is as follows:

`osjup70 dbname`

Replace *`dbname`* with the absolute or relative path of the database you want to upgrade.

You must upgrade each 6.*x* database that you want to use with 7.0. If you do not upgrade a database, and you try to use it with PSE Pro 7.0, PSE Pro does not open the old database and signals an exception.

osjversion: Obtaining PSE Pro Version Information

You can use the `osjversion` utility to display the version number and the build date for the version of PSE Pro you are running. This command is in the `bin` subdirectory of the installation directory. You must include `bin` in your path to run `osjversion`. For example:

```
C:\ODI\PSEProJ\bin>osjversion
ObjectStore PSE Pro for Java Release 7.0 for Windows NT Systems
Build packaged 04-22-07 23:11
```

This command is useful when you want to ensure that you have the right version in your path.

The `osjversion` utility checks what is available through the `PATH` environment variable. To check what is available through the `CLASSPATH` variable, you can write a program such as the following.

```
import com.odi.ObjectStore;
class OSVersion {
  public static void main (String[] args) {
    ObjectStore.initialize(null,null);
    System.out.println(ObjectStore.releaseName());
  }
}
```

Appendix
Packaging Your Application for End Users

When you package your application for delivery to end users, the package must include one and possibly two class files for each persistence-capable class in your application:

- The annotated class file
- The corresponding `ClassInfo` class file, if one was generated

For example, if you have a persistence-capable class called `Person`, in the `App` package, you must provide users with

- The annotated `App\Person.class` file
- The `App\PersonClassInfo.class` file, if present

There is no corresponding `ClassInfo` file for persistence-capable interfaces.

The class file postprocessor only generates `ClassInfo` files for persistence-capable classes if the class does not provide public access to all the required fields and methods. In most cases, `ClassInfo` instance is created as needed at run time. Check the destination directory specified for the postprocessor to determine which `ClassInfo` files were generated.

You can jar these files with the rest of your package. You need not send the unannotated version of your persistence-capable classes. The annotated version can be used in a transient context.

For persistence-aware classes, you must provide the annotated class file. There is no corresponding `ClassInfo` class file.

Glossary

active persistent object

An active persistent object starts as an exact copy of the object that it represents in the database. PSE Pro initializes a hollow object so that it becomes an active object. This happens when an application calls the `ObjectStore.fetch()` or `ObjectStore.dirty()` method. If an application calls the `ObjectStore.dirty()` method, rather than the `ObjectStore.fetch()` method, on a hollow object, the resulting active object can be modified.

Consequently, an active object is not necessarily identical to the object in the database that it represents. An application can read or update an active persistent object; a persistent object must be active for an application to read or update it.

API object

An object defined by the PSE Pro API that is associated with one session. An API object can be an object of one of the following classes: `Cluster`, `Database`, `DatabaseRootEnumeration`, `DatabaseSegmentEnumeration`, `Segment`, or `Transaction`.

batch

A batch is a set of files that must be postprocessed together. Often, this is all the files in your application. In more complex applications, there might be multiple batches that each contain a library and a batch of files that you write, which reference the libraries.

database

Persistent storage is organized into databases. Before a persistent object can be created, the database in which it is to be stored must exist, and this database must be opened by the process performing the creation. The database must also be opened by any processes accessing the object.

hollow persistent object

A hollow persistent object contains fields that are identical to the fields of the object in the database that the persistent object represents, but the fields have default values.

multistep index

An index on a complex navigational path that accesses multiple public data members, such as `a.b().c.name`.

persistence-aware

If the methods of a class can operate on persistent objects but an instance of the class cannot itself be stored in a database, the class is persistence aware.

When a method accesses fields in a persistent object, PSE Pro checks to ensure that the data has been read from the database. This checking is done by calls to the `ObjectStore.fetch()` and `ObjectStore.dirty()` methods. A persistence-aware class includes the annotations that call the `fetch()` and `dirty()` methods. It does not include the other annotations required

for a class to be persistence capable. Normally, you run the class file postprocessor to annotate a class so that it is persistence aware. Occasionally, you manually annotate the class yourself.

persistence-capable

A persistence-capable object has the capacity to be stored in a database. If you can store the instances of a class in a database, the class is persistence capable and the instances of the class are persistence-capable objects.

The definition of a persistence-capable class includes specific annotations required by PSE Pro. After you compile class definitions, you run the PSE Pro class file postprocessor on the compiled classes to add the annotations that make the classes persistence capable.

Some Java-supplied classes are persistence capable. Others are not persistence capable and cannot be made persistence capable. A third category of classes can be made persistence capable but there are important issues to consider when you do so. Be sure to read Java-Supplied Persistence-Capable Classes on page 261.

persistent object

A persistent object is a representation of an object that is stored in a database.

After an application retrieves an object from the database, the application works with the persistent object in the Java environment. A persistent object always exists in one of three states:

- Hollow
- Active
- Stale

session

A session allows the use of the PSE Pro API. PSE Pro uses the abstract `com.odi.Session` class to represent sessions.

Your application must create a session. After a session is created, it is an active session. A session remains active until your application or PSE Pro terminates it. After a session is terminated, it is never used again. You can, however, create a new session.

A session consists of a set of persistent objects and a set of PSE Pro API objects such as a `Database` and a `Transaction`.

In a single Java VM, PSE Pro and ObjectStore allow multiple concurrent sessions.

If you are using ObjectStore or PSE Pro, separate Java virtual machines can each run multiple sessions at the same time. See How Sessions Keep Threads Organized on page 37.

stale persistent object

A stale persistent object is no longer valid. Its fields have default values and it should not be used.

A persistent object might become stale after an application commits or aborts a transaction in which the active or hollow persistent object was accessible. When an application calls the `ObjectStore.destroy()` method, the target of the destroy method becomes stale.

An application must not try to read or update a stale object.

transitive persistence When an application commits a transaction, it stores in the database any transient objects that can be transitively reached from any persistent objects. This is the process of transitive persistence.