

# Ardent Java Bindings for O<sub>2</sub> and Relational Databases

Release 5.0 - May 1998

---



Information in this document is subject to change without notice and should not be construed as a commitment by O<sub>2</sub> Technology.

The software described in this document is delivered under a license or nondisclosure agreement.

The software can only be used or copied in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's own use.

Copyright 1992-1998 by O<sub>2</sub> Technology.

All rights reserved. No part of this publication can be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopy without prior written permission of O<sub>2</sub> Technology.

O<sub>2</sub> and O<sub>2</sub>Engine API, O<sub>2</sub>C, O<sub>2</sub>Corba, O<sub>2</sub>DBAccess, O<sub>2</sub>Engine, O<sub>2</sub>Graph, O<sub>2</sub>Kit, O<sub>2</sub>Look, O<sub>2</sub>Store, O<sub>2</sub>Tools are registered trademarks of O<sub>2</sub> Technology.

SQL and AIX are registered trademarks of International Business Machines Corporation.

Sun, SunOS and SOLARIS are registered trademarks of Sun Microsystems, Inc.

X Window System is a registered trademark of the Massachusetts Institute of Technology.

Unix is a registered trademark of Unix System Laboratories, Inc.

HPUX is a registered trademark of Hewlett-Packard Company.

BOSX is a registered trademark of Bull S.A.

IRIX is a registered trademark of Siemens Nixdorf, A.G.

NeXTStep is a registered trademark of the NeXT Computer, Inc.

Purify, Quantify are registered trademarks of Rational Software Inc.

Windows is a registered trademark of Microsoft Corporation.

All other company or product names quoted are trademarks or registered trademarks of their respective trademark holders.

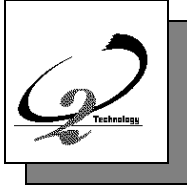
## **Who should read this manual**

This manual is for programmers who wish to write or adapt Java applications for the O<sub>2</sub> system and relational database systems. It presents the basic concepts of the O<sub>2</sub> Java binding, describes how to import classes and how to program using the interface.

Other documents available are outlined, click below.

**See [O2 Documentation set](#).**





# TABLE OF CONTENTS

---

The JB User Manual is composed of the following chapters :

- 1 [Introduction](#) to O<sub>2</sub> and Java Bindings.
- 2 [Getting Started](#) gives a general overview of the various components of JB and the different tasks of the development of a JB application, from database creation to application programming.
- 3 [Importing Java Classes](#) depicts how to make Java objects persistent capable by importing them into a database.
- 4 [Managing Persistent Java Objects](#) describes how the classes composing the JB API can be used to build applications.
- 5 [Advanced Programming](#) discusses performance tuning.
- 6 [JB Tools Reference](#) is a complete reference of the various JB tools provided in the current release.



---

# TABLE OF CONTENTS

---

<b>1</b>	<b>Introduction</b>	<b>11</b>
	<b>1.1 System overview .....</b>	<b>12</b>
	<b>1.2 Ardent Java Bindings Overview .....</b>	<b>14</b>
	Simple to use .....	16
	A first example .....	17
	Transparent to the user .....	19
	High performance .....	20
	Java application portability .....	21
<b>2</b>	<b>Getting Started</b>	<b>23</b>
	<b>2.1 Initializing JB databases.....</b>	<b>24</b>
	Initializing the O2 database.....	25
	Initializing the JB/JDBC database .....	25
	Initializing the JB/NATIVE database .....	25
	<b>2.2 Importing classes into a database.....</b>	<b>27</b>
	<b>2.3 Writing JB programs .....</b>	<b>30</b>
<b>3</b>	<b>Importing Java Classes</b>	<b>33</b>
	<b>3.1 Overview .....</b>	<b>34</b>
	<b>3.2 Importing classes.....</b>	<b>34</b>
	Importing classes of a package .....	34
	Confirming schema updates in O2.....	36
	Generated .class files.....	36
	Importing interfaces .....	37
	Importing arrays .....	37
	Re-importing .....	38
	Pre-imported classes .....	38
	<b>3.3 Unimporting classes .....</b>	<b>39</b>
	<b>3.4 The configuration file.....</b>	<b>40</b>
	Transient fields.....	41
	String fields .....	42
	Byte array variables .....	43

---

## TABLE OF CONTENTS

---

	Renaming.....	43
<b>3.5</b>	<b>Mapping Java types to O2 and SQL types .....</b>	<b>46</b>
	Java - O2 types .....	46
	Java - SQL types.....	47
<b>4</b>	<b>Managing Persistent Java Objects .....</b>	<b>49</b>
<b>4.1</b>	<b>Introduction.....</b>	<b>50</b>
<b>4.2</b>	<b>Database Management.....</b>	<b>51</b>
	Database Object.....	51
	Database Connection .....	52
<b>4.3</b>	<b>Transaction Management.....</b>	<b>53</b>
<b>4.4</b>	<b>JB Object States .....</b>	<b>55</b>
<b>4.5</b>	<b>Creating persistent objects .....</b>	<b>55</b>
	Example.....	56
	Array Instance Field .....	58
	String Type Instance Field.....	59
	Transient Instance Field .....	59
<b>4.6</b>	<b>Database Entry Points.....</b>	<b>60</b>
<b>4.7</b>	<b>Class Extent .....</b>	<b>60</b>
	OQL Predicate .....	61
	Referring to extents in OQL .....	62
	Join predicate.....	62
<b>4.8</b>	<b>Static fields.....</b>	<b>63</b>
<b>4.9</b>	<b>Traversing object references.....</b>	<b>63</b>
	Retrieving array fields .....	64
	Updating persistent objects.....	64
<b>4.10</b>	<b>Deleting persistent objects .....</b>	<b>65</b>
<b>4.11</b>	<b>Locking objects .....</b>	<b>66</b>
	Avoiding deadlocks .....	67
	Locking extents .....	68



---

## TABLE OF CONTENTS

---

	<b>4.12 Activate and prepareToWrite.....</b>	<b>69</b>
	activate .....	69
	prepareToWrite .....	69
	<b>4.13 Object identifiers .....</b>	<b>69</b>
	<b>4.14 JDBC driver facilities .....</b>	<b>70</b>
	<b>4.15 Exceptions .....</b>	<b>71</b>
	<b>4.16 User Administration.....</b>	<b>73</b>
	<b>4.17 User Access Administration .....</b>	<b>74</b>
<b>5</b>	<b>Advanced Programming .....</b>	<b>77</b>
	<b>5.1 Persistence in a non transparent way.....</b>	<b>78</b>
	Accessing a persistent object .....	79
	Updating a persistent object.....	79
	Persistent arrays .....	80
	String fields .....	83
	State transition of database events .....	83
	<b>5.2 Tuning Memory Management.....</b>	<b>85</b>
	Releasing objects from memory .....	85
	Writing objects to the database .....	86
	<b>5.3 Storing bytecodes in a database .....</b>	<b>87</b>
	<b>5.4 Using O2 schemas with Java applications .....</b>	<b>92</b>
	<b>5.5 Inspecting the system catalog .....</b>	<b>93</b>
	Inspecting imported classes .....	93
	Inspecting database structures.....	94
<b>6</b>	<b>JB Tools Reference .....</b>	<b>97</b>
	URL syntax .....	99
	o2jb_create_base.....	100
	o2jb_create_importer .....	103
	o2jb_create_schema .....	105
	o2jb_delete_base.....	106



---

## TABLE OF CONTENTS

---

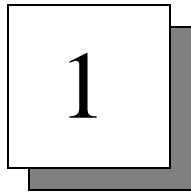
o2jb_delete_bytecode .....	108
o2jb_delete_importer.....	110
o2jb_delete_schema .....	111
o2jb_dump_bytecode .....	112
o2jb_dump_classes.....	114
o2jb_dump_schema.....	116
o2jb_export.....	118
o2jb_import .....	119
o2jb_init_base.....	123
o2jb_patch_class.....	125
o2jb_store_bytecode.....	126
o2jb_test_driver .....	128
o2jb_unimport .....	129



---

## TABLE OF CONTENTS

---



# Introduction

---

The Ardent Java Bindings provide the Java application developer with a transparent way to store and manipulate Java objects in object-oriented and relational databases.

This chapter presents an overview of the O<sub>2</sub> system and introduces the Ardent Java Bindings.

## 1.1 System overview

The O<sub>2</sub> system architecture is illustrated in [Figure 1.1](#).

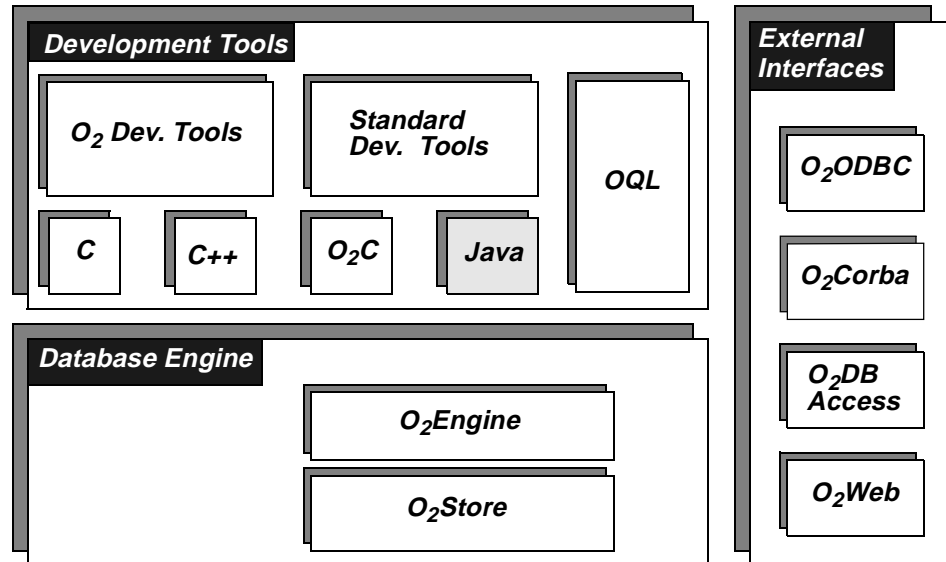


Figure 1.1: O<sub>2</sub> System Architecture

The O<sub>2</sub> system can be viewed as consisting of three components. The *Database Engine* provides all the features of a database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O<sub>2</sub> development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O<sub>2</sub> is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

### Database Engine:

- **O<sub>2</sub>Store** The database management system provides low level facilities, through O<sub>2</sub>Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
- **O<sub>2</sub>Engine** The object database engine provides direct control of schemas, classes, objects and transactions, through O<sub>2</sub>Engine API. It provides full text indexing and search capabilities with O<sub>2</sub>Search and spatial indexing and retrieval capabilities with O<sub>2</sub>Spatial. It includes a Notification manager for informing other clients connected to the same O<sub>2</sub> server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O<sub>2</sub> system.

---

## System overview

---

### Programming Languages:

O<sub>2</sub> objects may be created and managed using the following programming languages, utilizing all the features available with O<sub>2</sub> (persistence, collection management, transaction management, OQL queries, etc.)

- C O<sub>2</sub> functions can be invoked by C programs.
- C++ ODMG compliant C++ binding.
- Java Ardent Java Bindings.
- O<sub>2</sub>C A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O<sub>2</sub> objects and methods.

### O<sub>2</sub> Development Tools:

- O<sub>2</sub>Graph Create, modify and edit any type of object graph.
- O<sub>2</sub>Look Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O<sub>2</sub>Kit Library of predefined classes and methods for faster development of user applications.
- O<sub>2</sub>Tools Complete graphical programming environment to design and develop O<sub>2</sub> database applications.

### Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

### External Interfaces:

- O<sub>2</sub>Corba Create an O<sub>2</sub>/Orbix server to access an O<sub>2</sub> database with CORBA.
- O<sub>2</sub>DBAccess Connect O<sub>2</sub> applications to relational databases on remote hosts and invoke SQL statements.
- O<sub>2</sub>ODBC Connect remote ODBC client applications to O<sub>2</sub> databases.
- O<sub>2</sub>Web Create an O<sub>2</sub> World Wide Web server to access an O<sub>2</sub> database through the internet network.

## 1.2 Ardent Java Bindings Overview

---

Java is an object-oriented programming language. Therefore programming in Java naturally lends itself to an object-oriented database environment. The database allows you to store Java objects beyond the execution of your program. For system development and data management, such an object repository is essential. The Ardent Java Bindings offer you the optimal solution, providing transparent communication between your Java program and the database, portability of your Java programs to/from other database systems (relational such as Oracle, Sybase) without need of recompilation and tools to optimize data management.

Storing application data on persistent support (like hard disks) is a fundamental needs for all applications and also Java applications. Java environment proposes already different ways to store data, but all these solution have drawbacks:

- serialization for storing object on disk: can be used only with a small number of object, does not provide features like transaction, multi-users concurrent access, recovery,...)
- JDBC for storing object in relational databases: once more again, you hit the 'impedance mismatch' wall.

As opposed, the Ardent Java Bindings solves these problems by providing such features as multi-user concurrent access and query language in a powerful and easy way.

The Ardent Java Bindings allow you to store and retrieve Java objects as true objects without deriving from a specific base class or making explicit calls for storing and retrieving object. This binding itself is completely written in Java. As well, in an O<sub>2</sub> database, you can access any object independent of the programming language, C++ or other, used to create the object.

The Ardent Java Bindings allows your Java program to connect to any O<sub>2</sub> or relational databases. A Java application using JB, utilizes the underlying database as a transparent persistent object repository. Hence, the same Java application may run on many database systems without the need for any modification of the source code nor any recompilation. The figure below illustrates the general JB architecture.

One most powerful feature of the Ardent Java Bindings is the use of Java bytecode postprocessing. Persistence of Java object is realized not by modifying your Java source code, but by automatically inserting code in

---

## Ardent Java Bindings Overview

---

the bytecode generated by the Java compiler. This approach has many advantages:

- ease of debugging: the Ardent Java Bindings does not modify your source code.
- no needs of source code: you can make persist object from commercial package for which you don't have the source.

For optimizing purpose, you can also disable the postprocessing for a class, and manage yourself the reading and writing of objects directly in your code.

The Ardent Java Bindings use three ways for connecting to a database: for the O<sub>2</sub> database, for any JDBC-compliant database and for Oracle and Sybase databases. The first implementation runs on top of O<sub>2</sub> Engine, thus permitting maximum efficiency since O<sub>2</sub> is object-oriented. The JDBC implementation runs on top of any JDBC driver, and generates a relational schema from Java classes and the methods needed to read and write Java objects through JDBC. The third way is similar to the previous one but for performance optimizations, uses directly the client library of Sybase and Oracle databases.

The Java Binding Application Programming Interface (JB API) is implemented by a set of classes in the package `com.ardentsoftware.jb.api`. Various auxiliary tools are provided to allow the user to set up new databases and perform common administration tasks.

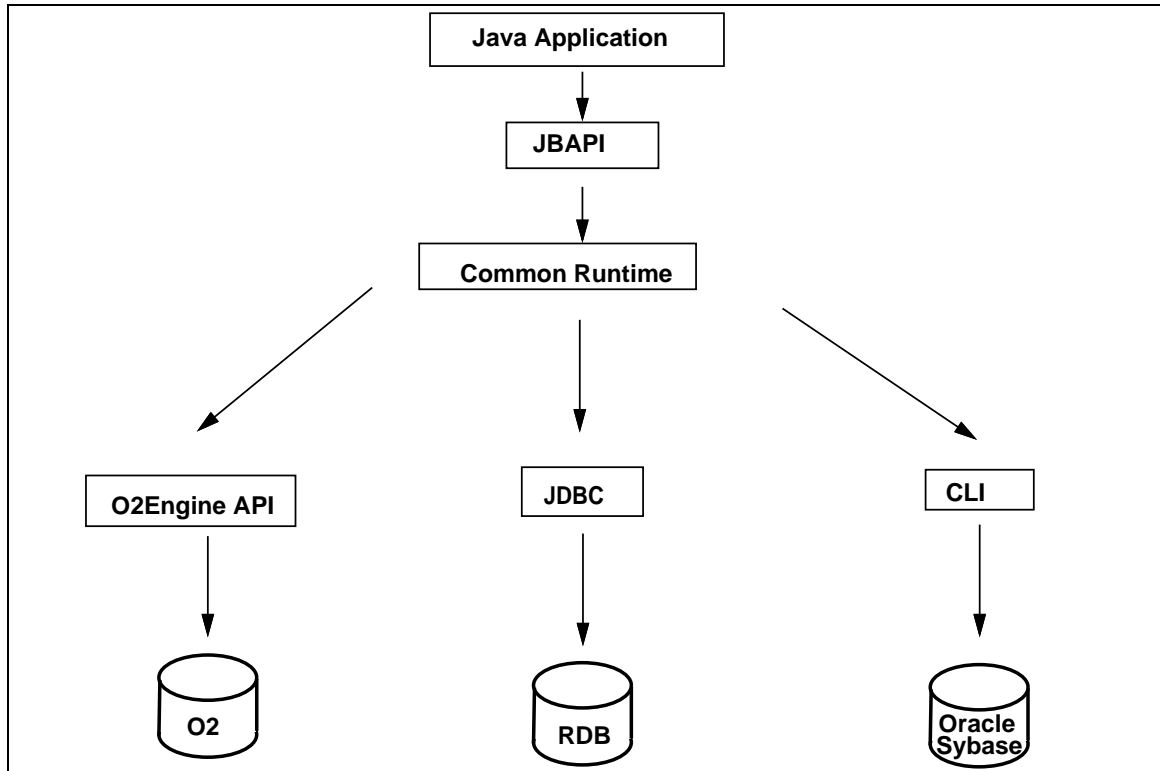


Figure 1.2: General Java Bindings Architecture

## Simple to use

JB provides a very simple interface to a database. It consists of a complete set of basic tools which allows you to develop Java applications that may access and manipulate persistent Java objects in an underlying database or any object in an O<sub>2</sub> database. Developing a JB application involves three distinct steps:

- initializing the underlying database server, where Java objects and class information are to be stored;
- importing Java classes to the database; this generates the database schema from the Java schema and modifies the Java class files with appropriate methods to manage persistence;
- coding applications using the JB API.



---

## Ardent Java Bindings Overview

---

In order to store a Java object in a particular database, you must first initialize the database and establish access privileges. Specific tools are provided for such administrative tasks.

To be able to store a Java object in a database, you must first import its class into the database. Any Java class can be imported to a database schema. In addition, any field of a Java class can be imported regardless of its visibility, except for transient and final fields which are not imported. Any Java object can therefore be partially or entirely stored in a database.

The JB API transaction provides a framework to make Java objects persistent and update such objects in your database with all the benefits of a transactional system in a multi-user environment. Stored objects can be retrieved directly using the corresponding class extents or by following a link from another stored object.

### A first example

The example below illustrates the three steps to develop a JB application; how to create a database, import a class and make persistent, and access a Java object from the database.

The first step is to initialize the database. After launching an O<sub>2</sub> server for a system `mySystem` on a machine `myServer`, create your schema and base as follows:

```
o2jb_create_schema -url o2:myServer:mySystem -schema schema1
o2jb_create_base -url o2:myServer:mySystem -base base1
```

For our example, we chose a simple `Person` class, composed of a `name` string and integer `age`. Compile `Person.java` with `javac` to obtain the `Person.class` file.

```
public class Person {
    private String name;
    private int age;
    public Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Note that any Java class can be made persistent in a transparent way without incorporating any JB API code into the class definitions.

The second step is to import the desired classes to the database. In our example, the class `Person` is imported to the database schema `schema1`. A URL is used to identify the database system. For `O2`, a user and password parameters are not required, as are for Oracle and Sybase. With `O2`, you must confirm any changes to the schema before opening a base.

```
o2jb_import -url o2:myServer:mySystem -confirm
            -schema schema1 Person
```

The third step is to use the JB API to access and modify the database. In our example, the class `example` containing the `main` is the application.

After importing a class, we can access and store objects in the database. In our example, a new person, `Jane`, is added to the database. This is carried out within a transaction, where the method `persist` marks the object to be written in the database. Only at commit time though, is the database actually changed. In this way, an abort can be used if you decide not to write to the database and end the transaction.

Within a separate session the database `base1` is accessed and queried. Using the class `Extent`, the database is queried for the number of persons over a certain age. The class `Extent` returned all the instances that satisfied the query, and using the `size` method provides us with the result.

Of course, you may query for a particular person and modify the information in the database, or simply use the information for other purposes.

---

## Ardent Java Bindings Overview

---

```
import com.ardentsoftware.jb.api.*;
import Person;

public class example {
    public static void main(String[] args)
        throws DBException {

    Person person = new Person("Jane",35);
    Database database = new Database("o2:myServer:mySystem","","");
    database.connect();
        database.open("base1");
            Transaction transaction = new Transaction();
            transaction.begin();
            Database.persist(person);    // make Jane persistent
            transaction.commit();
        // the database is queried for instances of persons over age 36
        Extent ext = Extent.all("Person").
            where("this.age > 36");
        int no_person = ext.size();
        database.close();
    database.disconnect();
    System.out.println("The number of people older than 36 is: "
        + no_person);
    }
```

### Transparent to the user

The Ardent Java Bindings allow objects to be stored and retrieved from an underlying database in a transparent way. In other words, you do not need to deal with the underlying database system but can rely entirely on the Ardent Java API when developing your applications. The Ardent Java Binding offers a high level interface which is transparent to the user, as opposed to JDBC for instance, which obliges you to manage the conversion between Java objects and relational tables.

In addition to Java objects, JB also allows you to store Java class bytecode. A class is provided which can load class bytecode directly from a database into a Java runtime system.

Stored Java objects can be retrieved directly using their corresponding class extent. A class *extent* collects all persistent objects of a given

class. A *proper extent* collects persistent objects only in a given class, while an *all extent* collects all persistent objects both in the given class and all subclasses. Objects are loaded on demand through class extents. A class extent implements an iterator on the corresponding structure in the database. It is possible to associate a predicate to the class extent and retrieve objects based on their contents, in a **select-from-where** style. As well, you can define data entry points to the database by using static fields.

### High performance

Java objects read from or written to the database are kept in a client cache to improve performance. Using a cache for data on the client side reduces the database and network traffic and improves performance by orders of magnitude. By separating the import phase from the database interface runtime, the performance of the latter is considerably improved. At runtime, all database structures required to make Java objects persistent are already installed. This reduces the number of checks performed at runtime compared with a dynamic import approach.

Other techniques are applied to optimize the transfer of data from/to the database, i.e. reading/writing of objects. For instance, for relational databases, stored procedures generated when a class is imported are used to perform insertion, deletion and update of data in the underlying database. You can pre-load/pre-dump objects pointed to by an object being read/written. As well, special data types are used to speed up the storage and retrieval of strings and array variables.

The performance of the Ardent Java Bindings can be tuned to meet particular application needs. In general these facilities should not be necessary, but can be applied in data intensive applications with special space and time related requirements. For example, if a stored object is no longer needed by the application in the current database session, the programmer can delete the object from the database cache (not from the database) which allows it to be garbage collected. Such a facility can be used to reduce and optimize memory space usage. Another facility for tuning memory space usage concerns array fields. Special methods are provided to allow array elements to be loaded upon demand from the database. This avoids having to load an entire array when only a few elements are needed.

---

## Ardent Java Bindings Overview

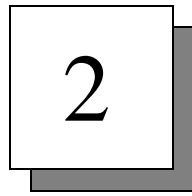
---

### Java application portability

In order to allow automatic portability and fully benefit from the security features of the Java programming language, the Ardent JB was implemented in Java. The version based on JDBC is 100% Java. For other implementations, a minimal amount of C code is used when absolutely necessary. The Ardent JB is implemented with the standard Java 1.1 platform. The implementation uses standard Java classes where appropriate and builds on and reinforces the style and virtues of the existing core Java classes. For example, errors and exceptions raised by the underlying physical database are captured by the runtime system and transmitted to the Java application program as a Java exception. Internal structures of the database runtime system, such as the database cache, are implemented with standard Java utility classes (e.g. `Hashtable`).



---



# Getting Started

---

Developing Java applications with Ardent Java Bindings involves three steps: initializing the database, importing Java classes to build the database schema, and writing JB-compliant Java applications. In this chapter, we introduce the main JB concepts and present an overview of the way a JB application is developed. Details of the different concepts and tools introduced here are provided in later chapters.

This chapter contains the following sections:

- *Initializing JB databases*
- *Importing classes into a database*
- *Writing JB programs*

## 2.1 Initializing JB databases

---

JB works with various database systems, for example: O<sub>2</sub>, Oracle, Sybase, MS-SQL or any relational database accessible through a JDBC driver. Since each database does not provide the same functionality, some JB tools and options are specific for certain databases. Any JB functionality specific to a particular database is described in this manual.

The following notation is used when referring to specific databases :

- JB/O<sub>2</sub> version: JB used with the O<sub>2</sub> database system
- JB/JDBC version: JB used with a JDBC driver
- JB/NATIVE or JB/rdbname version: JB used directly with a relational product client library. For example JB/ORACLE is the version of JB using the OCI library.

For a particular JB command, you specify the type of database to use with a database connection parameter as a form of URL. The URL is a string of tokens separated by a colon (":") character. The first token specifies the type of database system. Valid values are:

- ***o2***
- ***jdbc***
- ***oracle***
- ***sybase***

For O<sub>2</sub>, the second token is the name of the computer on which the o2server is running. The third token is the name of the O<sub>2</sub> system you want to connect to. For example:

```
o2:myServer:myJavaSystem
```

For Oracle, the URL is simply : **oracle**

For Sybase, the second token specifies the name of the sybase system, for example:

```
sybase:myJavaStore
```

For JDBC, the syntax follows the recommended URL for the particular JDBC drivers. For example:

```
jdbc:odbc:myDataSource
```



---

## Initializing JB databases

---

### Initializing the O<sub>2</sub> database

As explained in the [O<sub>2</sub> System Administration Guide](#), objects of classes are created in O<sub>2</sub> within schemas. Before importing Java classes you must create a schema by using the tool `o2jb_create_schema`, detailed in Chapter 6, Section [o2jb\\_create\\_schema](#).

The next step is to create an O<sub>2</sub> base, by calling the `o2jb_create_base` tool, detailed in Chapter 6, Section [o2jb\\_create\\_base](#). Alternatively you may create schemas and bases using standard O<sub>2</sub> DSA and O<sub>2</sub> DBA commands (see both *O<sub>2</sub> Administration* manuals).

```
o2jb_create_schema -schema mySchema -url o2:myServer:mySystem
o2jb_create_base  -schema mySchema
                  -url o2:myServer:mySystem -base myBase
```

Your database has now been initialized and Java objects may be made persistent in your O<sub>2</sub> base.

You can also access an O<sub>2</sub> schema which was previously created, whether with Java, C++ or O<sub>2</sub>C. To use a Java application on top of such a schema you must load an O<sub>2</sub> supplied file `o2java.dump` and import it into your schema, see [Section 5.4, Using O<sub>2</sub> schemas with Java applications](#).

### Initializing the JB/JDBC database

You must first create your own database with the tools provided by your particular database system. In order to initialize the database with the JB API use the `o2jb_init_base` tool, detailed in Chapter 6, Section [o2jb\\_init\\_base](#).

The `o2jb_init_base` tool alters your relational database by creating certain tables used to manage the imported classes

### Initializing the JB/NATIVE database

#### ***JB user privileges***

JB/ORACLE and JB/SYBASE provide two levels of user privileges, a *JB importer* and a *JB programmer*. An importer has special rights to import classes into a database, i.e. to make Java classes persistent capable. A

programmer writes applications using previously imported classes and the JB Application Programming Interface (API).

A user with administration privileges can specify a JB importer using the `o2jb_create_importer` tool, detailed in Chapter 6, Section [o2jb\\_create\\_importer](#).

The following example specifies the user `sa` with the password `sapasswd` as a JB importer on a Sybase database.

```
o2jb_create_importer -admin sa -adminpasswd sapasswd
-user jb_sa -passwd jb_sapasswd -url sybase:mySybase
```

---

### Note

This tool does not exist for the JB/O<sub>2</sub> and JB/JDBC versions.

---

### Creating a JB partition

In order to store and retrieve data from a database, a JB partition must be created and initialized. The tools `o2jb_create_base` and `o2jb_init_base` create and initialize a JB partition respectively. Only a JB importer can invoke these tools. The example below describes how to create a Sybase database named `base1` of size 100 MB by the importer `jb_sa`, note that a device name, log file and size must be provided.

```
o2jb_create_base -user jb_sa -passwd jb_sapasswd -base base1
-size 100 -url sybase:mySybase -device mydevice
-log mylog -logsize 6
o2jb_init_base -user jb_sa -passwd jb_sapasswd
-base base1 -url sybase:Sybase
```

The argument `-size` initializes the physical volume of the database partition to the given size in MB. Similarly, the argument `-logsize` initializes the log file to the given size in MB. A volume can be further extended by using administration tools provided by the corresponding

---

## Importing classes into a database

---

underlying database system. Details on the `o2jb_create_base` tool are provided in Chapter 6, Section [o2jb\\_create\\_base](#). The importer which created a database partition becomes the *owner* of this database. The owner of a database is the only user authorized to import classes into the database and to grant access rights to other users, as illustrated in the next section.

### 2.2 Importing classes into a database

---

After creating and initializing a database, you may import Java classes into the database. By importing a Java class, it becomes *persistent capable*, thereby instances of this class can be stored in the database. Importing a class entails the installation of all schema structures necessary to store objects in the database.

The tool `o2jb_import` is used to import classes to a database. Note that you must compile a class with the standard Java compiler before importing it. Before creating a base, or opening a base in a modified schema, you must confirm the updates by using the `-confirm` option (For more information about the `-confirm` option refer to [Confirming schema updates in O2](#) in Section 3).

The import tool also automatically generates bytecode (transparent to the user) to implement the communication between the application and the database engine. This is carried out during a "post-processing" phase where bytecode is inserted to the Java class methods, which allows the management of persistent objects in a transparent way.

In the following example, the class `student`, a subclass of `Person`, is imported to the O<sub>2</sub> database.

```
package example;
public class Person {
    private String name;
    private int age;
    private Person spouse;
    private Person children[];
    public Person getSpouse() {...} // getSpouse method
    public Person[] getChildren() {...} // getChildren method
    public void setSpouse(...) {...} // setSpouse method
    public void setChildren(...) {...} // setChildren method
}
public class Student extends Person {
    private Person responsible;
}
o2jb_import -url o2:myserver:mssystem -confirm
            -schema myschema example.Student
```

To import the same class into a Sybase database `base1` of the Sybase server `java_store`, the importer `jb_sa` uses the following command :

```
o2jb_import -url sybase:java_store -base base1
            -user jb_sa -passwd jb_spasswd example.Student
```

After executing the above command, applications using the JB API can store instances of `Student` (and of `Person`) in the database.

---

### **Important**

---

Following the concept of encapsulation, attributes of persistent classes must be private to assure transparency for access and update of persistent objects. Otherwise you have to explicitly use the `access` and `markModify` methods of the JB API. More details on the management of persistent objects in a non transparent way are provided in [Section 4](#).

---

---

## Importing classes into a database

---

---

### **Note**

---

In the JB/O<sub>2</sub> version, the `-user` and `-passwd` options of `o2jb_import` are ignored. In the JB/JDBC version, the values of the `-user` and `-passwd` options must identify a user with table creation privileges.

---

With a JDBC driver, an import command for the above example could be:

```
o2jb_import -url jdbc:ff-sybase://bdlys:2500 -base base1
            -driverclass sybase.jdbc.Driver -user jb_sa
            -passwd jb_sapasswd example.Student
```

where the values `ff-sybase://bdlys:2500` are JDBC driver dependent; the supplier of the JDBC driver is declared by `ff-sybase` and the server is `//bdlys` using port `2500`.

### **Import process**

Figure 2.1 summarizes the entire import process. [Section 3](#) presents how to import Java classes. The configuration file which may be used in the import process is described in [Section 3.4, The configuration file](#). For details about the `o2jb_import` tool, see [o2jb\\_import](#) in [Section 6](#).

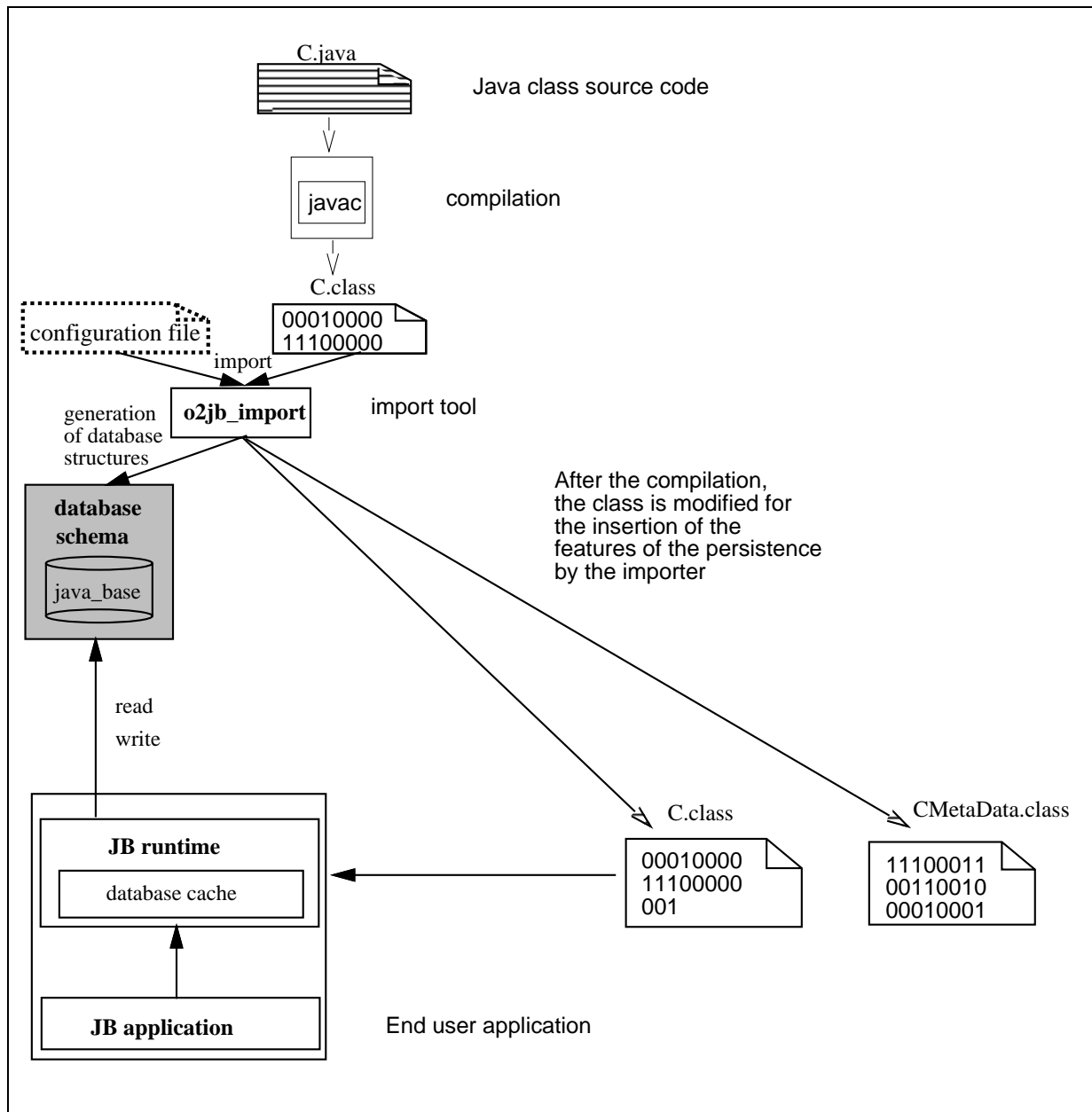


Figure 2.1 : The import process

## 2.3 Writing JB programs

The JB API is a set of classes composing the package `com.ardentsoftware.jb.api`. These public classes are used to

---

## Writing JB programs

---

develop JB applications and provide a high level interface to an underlying database where Java objects are stored.

The example below illustrates how to store instances of a class `Person` in a database. The classes `Database` and `transaction` defined in the `com.ardentsoftware.jb.api` package, are used to establish a connection with the database and start a transaction. Java objects are marked persistent with the method `persist`, and upon transaction commit (or validate) these objects are written in the database.

```
Person john = new Person("John",35),
      mary = new Person("Mary",32),
      bob  = new Person("Bob",5);
john.setSpouse(mary);
mary.setSpouse(john);
john.setChildren(0, bob); // children[0] = bob
mary.setChildren(0, bob);
Database database = new Database("database_URL",
                                "user","password");
database.connect();
  database.open("base1");
    Transaction transaction = new Transaction();
    transaction.begin();
    Database.persist(john);      // make John persistent
    Database.persist(john.getChildren(),0); // make the
                                        // ``children`` entry persistent
    Database.persist(mary);     // make Mary persistent
    Database.persist(mary.getChildren(),0);
    Database.persist(bob);     // make bob persistent
    transaction.commit();
  database.close();
database.disconnect();
```

The example below shows how persistent objects from the previous example can be accessed and updated in the database in a transparent way. The persistent object `bob` is accessed within a transaction to be

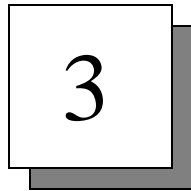
further updated, loaded into memory and written in the database in a transparent way.

```
Database database = new Database("database_URL",
                                "user", "password");
database.connect();
database.open("base1");
    // the object ``John`` is retrieved
    // by querying the Person extent:
Person john = (Person)Extent.all("example.Person").
    where("this.name=\"John\"").element();
Transaction transaction = new Transaction();
transaction.begin();
    Person bob = john.getChild(0);
    bob.setAge(6);
transaction.commit();
database.close();
database.disconnect();
```

The code in the above example is independent of any underlying database. Only the parameters of the database object constructor depend on the actual base. Therefore the same code can run on O<sub>2</sub> or any relational database without recompilation.



---



## Importing Java Classes

---

The Ardent Java Bindings allow you to store Java objects in a database, i.e. to make Java objects persistent. To store an object in a database, the object's class must be persistent capable. In this chapter, we describe how to import Java classes into a database to make them persistent capable. This chapter contains the following sections :

- *Overview*
- *Importing classes*
- *Unimporting classes*
- *The configuration file*
- *Mapping Java types to O2 and SQL types*

### 3.1 Overview

---

To store and retrieve objects of a Java class from a database you must first import the Java class to the database. Java classes are imported into a database using the `o2jb_import` tool. This tool generates and installs the database structures necessary to store and retrieve objects of a Java class from a database in an optimized and transparent way.

The import tool performs two major tasks. It generates methods to manage persistence of Java objects, during the *code generation* phase. And modifies user's methods during the *post-processing* phase. This last phase is called "post processing" because it occurs after the standard Java compilation. The source code of an application is not changed by the import tool, only Java `.class` files are patched. To summarize, patching a Java class consists of two tasks: insertion of byte code corresponding to methods that manage persistence (code generation) and altering programmer's methods when necessary by insertion of calls to methods that provide communication between the runtime and database ("post processing").

For each imported class `C`, the import tool also generates a file `CMetaData.class` which contains the description of the structure of the class. These generated data structures are used by the runtime to transfer data between an application and the database. Your Java application is independent of the underlying database structures that are generated when a class is imported since the import tool does not alter the user's class definition file `C.java`.

### 3.2 Importing classes

---

The `o2jb_import` tool, as its name suggests, imports classes into a database. The tool maps the Java class structure to a database structure. Hence your Java application remains independent of the underlying database. This section presents how to import Java classes into a database. For a complete description of this tool see [o2jb\\_import](#) in chapter 6.

#### Importing classes of a package

Before importing your Java classes to a database, you must create and initialize the database. This is performed with the `o2jb_create_base`

---

## Importing classes

---

and `o2jb_init_base` tools, as detailed in Chapter 6, sections [o2jb\\_create\\_base](#) and [o2jb\\_init\\_base](#) respectively.

For a database that has been previously created and initialized, only the database owner, the user that created and initialized the database, can import classes into a database.

---

### Note

---

In JB/O2 before using the tool `o2jb_create_base` you must create the schema with the tool `o2jb_create_schema`.

There are two ways to import Java classes ( $C_i$ ) from a package. Either import all the classes of the package automatically with the same import command :

```
o2jb_import ... -package <package name>
```

or import specific classes of the package :

```
o2jb_import ... <package name>.C1 <package name>.C2
                <package name>.C3 ...
```

With JB/O2, for example:

```
o2jb_import -url o2:myserver:mssystem -confirm
            -schema mySchema example.Student
```

---

### Note

---

In JB/O2, you cannot import to the same schema, classes having the same name from different packages. You must import such classes to different O<sub>2</sub> schemas.

### Confirming schema updates in O<sub>2</sub>

The import tool creates or modifies classes in O<sub>2</sub> schemas. While designing a schema you can modify it as much as necessary. But before creating a base, or opening a base in a modified schema, you must confirm the updates.

For example you might create a schema, import classes, confirm the updates and then create a base. On the other hand, if you know you will modify the schema many times before creating a base, it is recommended not to confirm after each modification. Each confirmation causes the modification of the O<sub>2</sub> schema, creating a new version.

You can automatically confirm your classes using the `-confirm` option with the import tool. You can also confirm your classes using O<sub>2</sub>Shell or O<sub>2</sub>Tools.

See the *O<sub>2</sub> System Administration Guide* and *O<sub>2</sub> System Administration Reference Manual* for further details.

### Generated .class files

For each imported Java class `C`, the import tool patches the `.class` file, generated by the compilation of the Java source code. The resulting patched file `C.class` is stored either in the output directory given with the argument `-output` or in the current directory by default. In addition to the original Java source methods, this `C.class` file contains, the bytecode of methods that manage persistence and the appropriate calls to these methods. These calls assure the management of persistent objects in a transparent way if and only if all fields of the imported class are private and therefore accessed only through the patched methods. Otherwise the programmer must manage persistence explicitly using the appropriate JB API, see Chapter 4, [Managing Persistent Java Objects](#).

At import time, you are warned if any class field is not private.

Even if all class fields are private, the programmer can decide not to post process the class methods while importing, by adding the option `-noppp` with the `o2jb_import` command.

The import tool also generates a `.class` file corresponding to the `ClassMetaData` associated with the imported class. The `ClassMetaData` describes the structure of the imported class and is used by the runtime.

---

## Importing classes

---

### Importing interfaces

Java interfaces should not be explicitly imported into a database. That is, you should not use the tool `o2jb_import` with a Java Interface. Instead, every interface referenced to by an imported class is automatically imported. When an interface `I` is imported, a file `IMetaData.class` is automatically generated by the import tool.

### Importing arrays

In JB/O<sub>2</sub>, arrays are automatically imported without the user specifying anything.

In JB/JDBC and JB/Native, an array is automatically imported, if it is referenced to by an imported class. For example, if array `B` is imported and it inherits an array `A`, the latter is considered to be a super array and it is automatically imported. Nevertheless the class of the array elements must be imported explicitly. If you want to store a subarray or an array of objects which implements an interface, you must explicitly import the array with the `-array` option of the `o2jb_import` tool. Given the example below :

```
interface I{}
class A {
    I tabI[];
    A tabA[];
}
class B extends A {
}
class C implements I {
}
```

If you want to assign to `tabA` an array of `B` (`tabA = new B[10]`), i.e. array `A` is a super array and `B` the subarray, you must explicitly import the array of `B`:

```
o2jb_import ... -array B[]
```

If you want to assign an array of `C` (`tabI = new C[10]`), you must explicitly import the array of `C`:

```
o2jb_import ... -array C[]
```

### Re-importing

During the life of a database, the database schemas may evolve. Some modifications are logical, for example adding a new method to a class. Other modifications are physical, for example adding an attribute to a class.

Re-import facilities are necessary to cope with class evolution. In other words, the user must be able to re-import a class `C` into the database after it has been modified. If the class has been modified, the system checks for compatibility between the new structure of `C` and the one stored in the database in order to assure that any existing persistent instance of `C` (and any subclass of `C`) will not be lost. This means that the structure stored in the database **must be** a subset of the new structure. If the two structures are not compatible the class is not re-imported. In such a case the user can delete the class, and its associated objects (using the `o2jb_unimport` tool) before attempting to import it again.

Various database systems handle database evolution in different ways. The `O2` database accepts any changes in the structure of a class (see the [O<sub>2</sub> System Administration Guide](#)). A new schema can be imported at any time. For relational databases, only limited changes to the class structure are accepted.

A class update that does not entail any loss of stored instances is called a *conservative update*. The only conservative class update supported by relational databases, is the addition and deletion of a field in a class. All other class modifications (e.g. changing an attribute type, renaming an attribute or changing the class hierarchy) entail the loss of existing persistent instances in the database and are therefore not supported. Limitations of class evolution are due to restrictions of the underlying database system concerning structure modification.

### Pre-imported classes

JB provides a set of pre-imported classes which are :

---

## Unimporting classes

---

Class	Package
PersistentDate	com.ardentsoftware.jb.java.util
PersistentHashtable	com.ardentsoftware.jb.java.util
PersistentString	com.ardentsoftware.jb.java.lang
PersistentVector	com.ardentsoftware.jb.java.util

These classes are the persistent versions of the standard Java classes `Date`, `Vector`, and `Hashtable` of package `java.util` and `String` of package `java.lang`. A JB-compliant Java application may use these classes, without importing them into the database. A user's application can import classes which reference these three JB API classes but should not import them.

### 3.3 Unimporting classes

---

During the administration of a database it is useful to unimport a class. This can serve to delete a particular object from a database. As well the evolution of a class may require the deletion of a class before reimporting it. The `o2jb_unimport` tool is provided to delete (unimport) a class from a database.

In JB/O<sub>2</sub> you may delete (unimport) any class, even if it is referenced by another class or if the class has instances in the database. However you can not delete a class that has subclasses, but you can unimport the entire hierarchy with the same `o2jb_unimport` command.

In JB/JDBC and JB/NATIVE, a class may be deleted only if there are no stored instances of this class and if it is not referenced by another class, unless the referencing class is itself also being deleted with the same command. Subclasses of a deleted class are also automatically deleted. A complete description of the `o2jb_unimport` tool is given in Chapter 6, section [o2jb\\_unimport](#).

Below, the `o2jb_unimport` tool is used to delete from schema `myschema` the class `student` of package `example` that was

previously imported :

```
o2jb_unimport -url o2:mysqlserver:mssystem
               -schema mySchema example.Student
```

For JB/JDBC, the command is:

```
o2jb_unimport -url sybase:mySybase -base base1 -user jb_sa
               -passwd jb_spasswd example.Student
```

---

### Note

For JB/JDBC, the values of the `-user` and `-passwd` options must identify a user with table creation privileges.

---

## 3.4 The configuration file

---

JB provides a configuration file which allows you to override the default import options. Using this configuration file you can select class fields that are to be stored in the database and/or provide information so that the database system can optimize the storage of string values. In addition, you can rename classes and fields to generate different names in the database schema, as explained below.

The `o2jb_import` tool can read a configuration file, which you use by adding the `-config` option to your `o2jb_import` command.

The various options of the configuration file are described below:



---

## The configuration file

---

### Transient fields

The user can decide not to store certain fields of an object in the database. This is particularly useful when a field is used to temporarily hold the result of a computation. Such a field can have its value recomputed and is not stored in the database.

The syntax for declaring transient variables is as follows:

```
TRANSIENT ClassName: FieldName{,FieldName};
```

where **ClassName** is a (fully qualified) Java class name, which must be followed by a non empty list of comma-separated field names, for fields declared locally in class **ClassName**.

You can also use the standard Java qualifier **transient** in the Java class declaration. Fields declared as **transient** in a configuration file or in Java classes are not imported, i.e. they are not mapped to a corresponding structure in the database and are therefore not stored in the database.

Because a **static transient** declaration is invalid in the Java language, you must declare transient static fields as **transient** in the configuration file.

Another way to ignore static fields when importing is to use the **-nostatic** option of the import tool. This option automatically defines all the static fields of the class to be imported as **transient** fields.

A **final** field is by default transient and is never imported to a database, and is therefore ignored by the import tool.

---

#### Note

The Java keyword **transient** is also used with *object serialization*. In case of conflict, you must use a configuration file to declare transient variables.

---

#### Note

The method **activate** called by the runtime when a persistent object is loaded to memory can be used to initialize transient fields (see [Section 4.12](#)).

### String fields

JB can be used with various databases which support different default string lengths. You can use the configuration file to control the string length used for the various Java fields.

A configuration file can be used to redefine the default string type mapping and default string type size for the JB/JDBC and JB/NATIVE versions. This directive is ignored by the JB/O<sub>2</sub> version, since with O<sub>2</sub> databases there are no constraints neither on string lengths nor on the maximum size for indexing.

The import tool maps Java fields of type `string` to table columns of type `varchar`, of the maximum size provided by the underlying database. Therefore, such columns can hold strings of a limited size. If a given field is intended to store long strings, the default type can be redefined as a `long varchar` type to be used instead of `varchar`. As well, such columns may be too long for an index. If a given column is intended to be indexed, the default size may be redefined to be less than the maximum value permissible by the underlying database system.

The syntax for redefining string types is as follows:

```
TEXT ClassName: fieldName{,fieldName};
```

where `ClassName` is a (fully qualified) Java class name, which must be followed by a non-empty list of comma-separated field names, for fields of type `java.lang.String` declared locally in the class `ClassName`.

For an array of strings, it is not possible to redefine the type such that they are stored as a `long varchar` rather than a `varchar`.

The configuration file can be used to define the size of the `varchar` length to be less than the default size. The syntax for redefining `varchar` size is as follows:

```
VARCHAR size ClassName: fieldName{,fieldName};
```

where `ClassName` is a (fully qualified) Java class name, which must be followed by a non-empty list of comma-separated field names, for fields of type `java.lang.String` declared locally in the class `ClassName`. The option `size` is the size of the `varchar` field in the database.

---

## The configuration file

---

### Byte array variables

Java provides the byte data type that allows you to store binary files such as images, audio, etc.

For the JB/JDBC and JB/NATIVE versions, Java fields of type `byte[]` are by default mapped into database structures, where each array element is stored as a row of a table. With JB/O<sub>2</sub>, the `byte[]` array is stored as a `list(char)` structure. This default can be redefined so that arrays of bytes are stored in columns of a long varbinary type for the JB/JDBC and JB/NATIVE versions, or stored as `bits` for JB/O<sub>2</sub>.

The syntax for redefining byte array types is as follows:

```
BINARY ClassName: fieldName{,fieldName};
```

where `ClassName` is a (fully qualified) Java class name, which must be followed by a non-empty list of comma-separated variable names, for variables of type `byte[]` declared locally in the class `ClassName`.

### Renaming

The mapping of Java classes, possibly from various packages to a single database, may pose a problem in terms of conflicts between names used in Java for classes and fields. JB provides an automatic renaming mechanism to solve such conflicts. As well, the length of a name may be limited by the underlying database system.

With O<sub>2</sub> databases there are no constraints on identifier name lengths, and therefore no need for renaming, which is ignored by the JB/O<sub>2</sub> version. Though with JB/O<sub>2</sub>, you cannot import to the same schema, classes having the same name from different packages. You must import such classes to different O<sub>2</sub> schemas.

For the JB/JDBC and JB/NATIVE versions, classes are imported as tables; and fields are imported as columns in the corresponding tables. Tables and columns are named after the non-qualified names of the classes and fields from which they were derived.

A name conflict occurs when two classes having the same name from different packages are imported into the same database. The import tool solves such conflicts with automatic renaming. For instance, if multiple `Person` classes are imported from different packages into the same

database, the first one to be imported corresponds to the table **Person**, the second one to **Person2**, and so on.

Also, if the name of a class is a reserved word for the database system or class names are greater than the maximum size for identifiers allowed by the underlying database, an automatic renaming is performed. The keywords and maximum size of an identifier is database dependent, hence the name of a mapping table may differ in two different databases.

Automatic table renaming is reported to the user at import time (if the option `-verbose` of `o2jb_import` is set) and does not have any impact on the application, since knowing the real table name is not necessary when using the JB API, unless the user wants to reference a table name in the **where** clause of an extent. In that case, the application must be provided with the name of the table for the given class before using it in a query (see example in [Section 4.7](#)).

Because automatic renaming of column names has an impact on the use of the **where** clause of an extent, no automatic renaming is performed on field names and an exception is raised by the `o2jb_import` tool if the name is too long or if it is a key word.

You can solve name conflicts by renaming classes and fields using a configuration file. The syntax for renaming a class is as follows:

```
TABLE ClassName = TableName;
```

where **ClassName** is a (fully qualified) Java class name, and **TableName** is the name of the table to be generated for the class **ClassName**.

A new name, **TableName**, declared by the user or generated by the import tool through automatic renaming is used by the importer tool to generate the name of the import file instead of the original class name.

A new name provided for a table or column should not conflict with the name of any existing table or column in the same database; otherwise an exception is raised by the import tool.

---

### Note

---

The renaming of fields is not recommended because it has an impact on the way you write the **where** clause of an **extent**.

---

---

## The configuration file

---

The following example shows a configuration file for the imported class `Person`.

```
public
class Person {
    private static byte music[]; // binary in config file

    private int age;    // transient attribute in config file
    private String cv; // text attribute in config file
    private String name;
    private Person spouse;
}
```

An example configuration file for the class `Person` above. You can write comments in a configuration file following the number sign (#) symbol :

```
# transient field
TRANSIENT Person: age;
# This is a field with a large text value
TEXT Person: cv;
# Stores the music
BINARY Person: music;
```

### 3.5 Mapping Java types to O<sub>2</sub> and SQL types

#### Java - O<sub>2</sub> types

Java type	O <sub>2</sub> type
<i>Built-in types and corresponding classes</i>	
boolean	boolean
char	char
byte	char
short	integer
int	integer
long	two attributes of type integer, whose names are suffixed by hi and lo
float	real
double	real
String (UNICODE)	string (UTF)
<i>Constructed types</i>	
interface I {...}	class I (no data)
interface J extends I {...}	class J inherit I (no data)
class C {...}	class C public type tuple(...)
class C extends S implements I {...}	class C inherit S, I public type tuple (...)
<i>Java Type</i> []	class o2_list_ <i>O2Type</i> public type list( <i>O2Type</i> )
<i>Java Type</i> [], where class <i>JavaType</i> extends S implements I...	class o2_list_ <i>O2Type</i> inherit o2_list_S, o2_list_I public type list( <i>O2Type</i> )
<i>JavaType</i> [] []	class o2_list_o2_list_ <i>O2Type</i> public type list(o2_list_ <i>O2Type</i> )
<i>Special cases</i>	
byte[]	bits if the Java field is specified as a binary in the configuration file used when importing the Java class (see <a href="#">Section 3.4</a> )
<i>Static variables</i>	
in <i>class</i> , static <i>Java Type att</i>	name o2_var_ <i>class</i> : tuple ( <i>att</i> : <i>O2Type</i> ...)

---

## Mapping Java types to O2 and SQL types

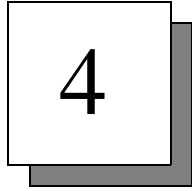
---

### Java - SQL types

Java type	SQL data type
byte, Byte	tinyint
short, Short	smallint
int, Integer	int
long, Long	double precision
float, Float	real
double, Double	double precision
char, Character	char
boolean, Boolean	bit
string	Varchar/Long Varchar
byte[]	Long Varbinary if the java field is specified as a binary in the configuration file used when importing the java class (see <a href="#">Section 3.4</a> )







# Managing Persistent Java Objects

---

The Ardent JB Application Programming Interface is a set of classes composing the package `com.ardentsoftware.jb.api`. These are public classes which allow to manage persistent Java objects. They provide a high level interface to the underlying database.

This chapter is composed of the following sections:

- *[Introduction](#)*
- *[Database Management](#)*
- *[Transaction Management](#)*
- *[JB Object States](#)*
- *[Creating persistent objects](#)*
- *[Database Entry Points](#)*
- *[Class Extent](#)*
- *[Static fields](#)*
- *[Traversing object references](#)*
- *[Deleting persistent objects](#)*
- *[Locking objects](#)*
- *[Activate and prepareToWrite](#)*
- *[Object identifiers](#)*
- *[JDBC driver facilities](#)*
- *[Exceptions](#)*
- *[User Administration](#)*
- *[User Access Administration](#)*

## 4.1 Introduction

---

The `com.ardentsoftware.jb.api` package is delivered as a set of `.class` files stored in the `o2jb.jar` file.

As a companion document you will find the full reference manual for the API in the HTML files; `$O2HOME/help/java` for JB/O2. The JB runtime interface follows the standard Java programming style. Therefore you use a single language to implement objects in memory and in the database. Hence the programmer does not deal with the complexity of two separate languages, as is the case with embedded SQL approaches like JDBC.

A JB application connects and opens a database through the methods `connect` and `open` of the class `Database`. A transaction is started by calling the method `begin` of the class `Transaction`. A transaction ends when the methods `commit`, `validate` or `abort` are called.

A Java object becomes persistent by calling the method `persist(Object)` of the class `Database`. Accessing persistent Java objects and updating them is as simple as accessing and updating transient objects.

A typical JB program has the following general structure:

```
database = new Database(...);
// create a database object with appropriate parameters
database.connect(); // open a connection to the database server
database.open("base1");
// read objects stored in the database in read-only mode
...
transaction = new Transaction();
transaction.begin(); // enter transaction mode
// create new objects in the database
... // update objects in the database
// delete objects from the database
transaction.commit();
// write persistent objects to the database
database.close(); // close the currently active base
database.open("base2"); // work on objects in base2
...
database.close: // close the currently active base
database.disconnect();
// close the connection to the database server
```

---

## Database Management

---

### 4.2 Database Management

---

#### Database Object

A database must be opened before being accessed and closed at the end of the database session. To open a database, a **Database** object must be created with three parameters: URL, user name and user password. To set these parameters use the **Database** constructor, as illustrated below.

```
database = new Database(URL,User_Name,User_Passwd);
```

For example:

```
database =  
new Database("DB_sys:my_server:java_store","john","john_pswd");
```

---

#### **Note**

---

The parameter **URL** has a different syntax according to the underlying system. The first part is always the type of system (o2, jdbc, sybase, oracle). The remaining parts are as follows :

- For JB/O<sub>2</sub>, the URL has the form "o2:server\_name:system\_name" where server\_name is the name of the o2server (i.e. the name of the computer on which the o2 server is running) and system\_name the name of the o2 system on which you want to work. The user name and the password are not taken into account.

```
database = new Database("o2:the_server:my_system", "", "");  
database.connect();  
database.open("my_base");
```

(For further details see the *O<sub>2</sub> System Administration Manual*.)

- For JB/O<sub>2</sub>, the URL is "o2:myserver:mssystem" where "mssystem" is the name of the O<sub>2</sub> system running on the machine called "myserver".
- For JB/JDBC, the URL has the same form as the URL defined in the JDBC standard. Before creating the **Database** object you must register the JDBC driver by calling the method **Class.forName**. For example, if you use the jdbc/odbc bridge, you must write:

```
// Registers the JDBC driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// Constructs the database object for the connection
database = new Database("jdbc:odbc:myDataSource",
                        "john","john_pswd");

database.connect();
database.open();
```

- For JB/Sybase, the URL has the form "sybase:system\_name", where system\_name is the name of the sybase database system.
- For JB/Oracle, the URL is simply "oracle" without anything else.

### Database Connection

The connection to the database server is established through the method **connect**. A connection must be established before attempting to open a given base. For example, to establish a connection with the server associated to the **Database** instance write:

```
database.connect();
```

To open the database, the method **open(String)** of the class **Database** is used, where string is the name of the base.

```
database.open("my_base");
```

---

## Transaction Management

---

If in the underlying database system there does not exist a base with the name passed as a parameter string, the exception `DBException` is raised.

At any time, only one base can be open. This base is called the current database. An attempt to open a base before closing the currently active database raises the exception `DatabaseAlreadyOpenException`. Several database objects can exist simultaneously, but transaction commands only apply to the current database.

When a base is opened, a database cache is created by the runtime system to handle data transfer to and from the underlying database. This cache contains handles to stored objects and is transparent to the user. When a database is closed, the current transaction, if any, is committed and the associated database cache is freed. The method `close` is used to close the current database, as follows:

```
database.close();
```

To disconnect from the server, the method `disconnect` is used.

```
database.disconnect();
```

For the complete `Database` class documentation, refer to the HTML document.

### 4.3 Transaction Management

---

By default, after opening a database, a Java application runs in a read-only transaction mode. In this mode, no updates to the underlying database are allowed. In other words, objects stored in the current database cannot be updated and new objects cannot be stored in that database. However this mode is the most efficient way to read objects in a multi-user environment, since it does not require to wait for any concurrent transactions to finish.

Loading persistent objects from the underlying database into memory (i.e. database cache) can be performed in read-only mode, whereas creation, modification and deletion of persistent objects must be performed within a transaction.

To be able to update persistent objects and create new persistent objects, a transaction must be explicitly started with the `begin` method of the `Transaction` class.

A transaction can be terminated in three different ways, through the `commit`, `validate` and `abort` methods.

When the current transaction is terminated with a `commit`, all updated objects are written to the database and modifications are committed, all locks are released and the database cache is cleared. If an updated object refers to a non-persistent object then the reference is set to null in the database. The method `validate` is similar to the method `commit`, but the JB cache is not cleared.

If the method `abort` is used, updated objects are not written to the database, all locks are released and the JB cache is cleared. As well, modifications are not validated in the database and objects declared persistent during the aborted transaction become transient (see below for details).

Once a transaction is terminated, the application runs in read-only mode.

When the database is closed, the current transaction is committed.

Nested transactions are not supported. If the `begin` method is called while a transaction is active, an exception (`ActiveTransactionException`) is raised.

After a `commit` or an `abort`, the cache is cleared and all handles to objects in the database are lost. After a `validate`, persistent objects remain in the database cache, and if accessed they are reloaded from the database in a transparent way.

As opposed to  $O_2$ , transaction or transaction isolation level 3 are not supported by some database systems, therefore JB cannot ensure the same behavior on these databases.

To obtain information about the level of concurrency provided by the underlying database, the method `getProperty` of the `Database` class is provided. This method returns a `JRBProperty` object. For information about transaction features, two methods of the `JRBProperty` class are provided: `hasTransaction()` and `hasIsolation3()`.

---

## JB Object States

---

For more details see [Section 4.14](#).

For the complete `Transaction` class documentation refer to the HTML documentation.

### 4.4 JB Object States

---

An object can be made persistent only if it is an instance of an imported class. Persistent objects can be created, loaded, modified and deleted from the database. An object in a JB application can be in one of the following states:

- ***transient***

A transient object is an object that has not been accessed from the database and exists only in the application program heap. It is therefore not in the JB cache and has no corresponding object stored in the underlying database.

- ***persistent***

A persistent object is an object that either has been marked as "`persist`" in the current transaction or has been accessed from the database in the current transaction. It has an associated handle in the JB cache and is therefore linked to an object stored in the underlying database. Persistent objects are used by an application like transient objects, on the condition that encapsulation of imported classes is respected, in other words, that all attributes of imported class are private. If this is not the case, refer to the chapter 5, "Advanced Programming" of this document.

### 4.5 Creating persistent objects

---

To make an object persistent you first create a transient object and then pass it as an argument to the method `persist(Object)` of class `Database`. Thus this object will be created in the database at commit or validate time.

JB supports a direct persistence model, i.e. objects made persistent are added to the corresponding class extent at commit time. Objects pointed to by a persistent object, however, do not become persistent until they are explicitly made persistent.

A Java object is composed of several "instance variables", whose types can be primitive (like integer), or string or reference to other objects or arrays.

A persistent object can refer to a transient object in memory but then at commit time the reference is set to NULL.

### Example

Let us consider an example based on class `Person` given below.

```
class Person {
    private int age;
    private String name;
    private Person spouse;
    public setSpouse(Person P){spouse=p;}
    public getSpouse() {return spouse;}
    ...
}
```

The following code writes an instance of `Person` to the database.

```
Person John = new Person("John");
Person Mary = new Person("Mary");
                // John and Mary are transient objects.
...
transaction.begin();

John.setSpouse(Mary);
Database.persist(John);    // John becomes persistent; it will
                            // be written in the database at commit time.
transaction.commit();     // John is written in the database
                            // Mary has not been made persistent,
                            // so John.spouse is set to null in the
                            // database, but not in memory.
...
}
```



---

## Creating persistent objects

---

In the previous example, the object `John` becomes persistent but `Mary` is still transient. To store the object `Mary`, it must be explicitly made persistent, as illustrated below.

```
...
transaction.begin();
John.setSpouse(Mary); // Mary is a transient object. John is persistent
Database.persist(Mary); // or Database.persist(john.getSpouse());
transaction.commit(); // Mary is also written in the database
```

Figure 4.1 depicts the persistent objects `John` and `Mary` as instances of the `Person` class. If `Mary` is not explicitly made persistent in the database, `John.getSpouse()` refers to null. As object `John` has been modified through a method (`setSpouse`), it is automatically updated in the database.

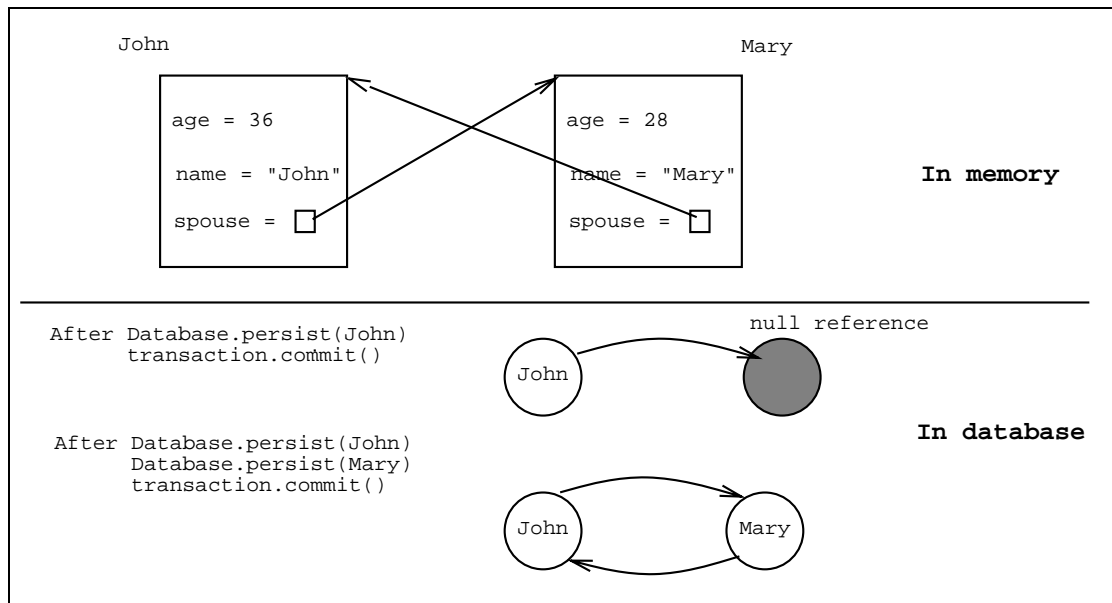


Figure 4.1 : Persistent objects `John` and `Mary`

### Array Instance Field

In Java an array is similar to an object, it must be created with a new operator.

For a persistent object, by default any instance field which is an array is not persistent. You must make the array explicitly persistent by calling the `persist` method of class `Database`.

If an array has references to objects, these objects must be made persistent explicitly, otherwise the references are set to NULL when the array is written in the database.

Example :

```
class Student extends Person {
    private int mark[];
    private Training trainings[];
    public Student (int msize, int tsize){
        mark = new int[msize];
        trainings = new Training[tsize];
    }
    public Training[] getTrainings() {return trainings;}
    public Training getTraining (int i) {return trainings[i];}
    public int[] getMark(){return mark;}
}
```

Let us create a `Student` persistent object.

```
transaction.begin();
...
Student Bob = new Student(3,2);
Database.persist(Bob);
transaction.commit();
```

In the database we have a `student` object with `mark` and `trainings` set to NULL. If we want those arrays to be persistent we must write :

---

## Creating persistent objects

---

```
transaction.begin();
Database.persist(Bob.getMark());
Database.persist(Bob.getTrainings());
transaction.commit();
```

Now the three integers of `mark` are stored in the database, but the two references to `Training` objects are still NULL. If we want to make persistent the `Training` objects we must write :

```
for (int i=0; i<Bob.getTrainings().length; i++)
    Database.persist(Bob.getTraining(i));
```

### String Type Instance Field

A string belonging to a persistent object is made automatically persistent. A NULL string is stored as an empty string in the database.

### Transient Instance Field

A transient instance variable is never stored in a persistent object. Transient fields are declared in the Java schema with the `transient` qualifier. Since Java does not support `static transient`, a configuration file can be used to declare a static attribute as detailed in [Section 3.4](#).

A configuration file should be used to declare transient fields of an object to avoid conflict when `transient` is used for *object serialization*. When a class is imported, fields declared as transient are not imported to the database schema.

Consider the example below. Field `age` in the class `Person` is intended to store the age computed from the field `birthDate`. As `age` is a derived attribute, we are not interested in storing its value for each persistent instance of `Person`.

```
class Person {
    private Date birthDate;
    transient int age; // It is thus not imported in the schema and is
                      // not stored in persistent instances of Person.
    private String name;
    private Person spouse;
    public void Person setAge (int age){...}; // method to set Age
}
...
Person p;
...
p.setAge(38);
Database.persist(p);
transaction.commit();    // the attribute age is not stored in the
                        // database.
```

The method `activate` described in [Section 4.12](#) allows to initialize a transient field, like `age`, when the object is loaded to memory.

## 4.6 Database Entry Points

---

A JB application can retrieve objects stored in a previous database session. Stored objects can be accessed through data entry points defined in the database. The system automatically creates entry points corresponding to *class extents*. User defined data entry points can also be defined by importing *static fields*.

## 4.7 Class Extent

---

A class extent contains all the instances of a class that have been explicitly updated, and written by a commit or validate operation. There are two kinds of class extents:

- *proper class extent*: only instances of the class are retrieved;
- *transitive class extent*: all instances of the class and its subclasses are retrieved.

---

## Class Extent

---

The class **Extent** is provided to allow retrieval of class extents for imported classes. A class extent can be filtered through a predicate. The syntax of a selection predicate is similar to that of the **where** clause of a **select-from-where** OQL query, which can also be used to query private fields of a class.

Example of class **Extent** :

```
Extent personExtent;
           // retrieve the transitive extent of class Person:
personExtent= Extent.all("Person");
           // retrieve the proper extent of class Person:
personExtent = Extent.proper("Person");
           // select a particular subset of instances of Person:
personExtent = Extent.all("Person").where("this.name = \"John\"
                                           and this.age = 36");
```

To enumerate the set of selected objects, an **Enumeration** object is returned by the **elements** method.

```
Person p;
for (Enumeration e = personExtent.elements() ;
     e.hasMoreElements() ;) {
    p = (Person)e.nextElement();
    ...
}
```

More elaborate queries are possible utilizing the class **Extent**.

### OQL Predicate

For the JB/NATIVE or JB/JDBC versions, the predicate associated with an extent is a standard SQL predicate. It can perform joins with other tables, given that the name of such tables are known.

In general, when the name of a table for a given class must be referenced in the selection predicate, this name should be retrieved through the method **getTableNameForClass** of class **Database**. Due to renamings, table names may be different from class names (see [Section 3.4](#)). The following example illustrates this :

```
String temp = database.getTableClassName("myPackage.Employee");
marriedToEmployee = Extent.all("myPackage.Person").
    where("spouse_oid in select oid from "+temp);
```

### Referring to extents in OQL

For the JB/O2 version, the name of the extent of a class *c* is always the persistent root `o2_extent_c`. The method `getTableClassName` of class `Database` has no meaning and returns null.

The predicate in the previous example can thus be written using `O2` as:

```
where("this.spouse in o2_extent_Employee);
```

### Join predicate

The `where` clause can refer to attributes which are references to other class objects. The method `oqlEquals` of class `Database` allows to test equality on such an attribute.

Example :

```
class Student {
    private Professor responsible;
}
Professor john;
john = Extent.proper("Professor").\
    where("this.name = \"John\").element();
...
Extent e = Extent.proper("Student").where("this.age = 21 and "\
    + Database.oqlEquals(john,"responsible"));
```

---

## Static fields

---

The last `where` clause is equivalent to the OQL query :

```
select s from s in Student
      where s.age = 21 and s.responsible=john
```

For the complete documentation of class `Extent` refer to the HTML documentation.

### 4.8 Static fields

---

Static fields provide a means for the user to define application-dependent roots of persistence. By default, at import time, a static field is set persistent and its value will be stored in the database. The methods managing persistence for static fields are generated by the import tool in `.class` files. These methods allow retrieval and storage of imported static fields from and to the database in a transparent way provided that static attributes are accessed or updated through methods of the imported class. You can override the default behavior by declaring the static field transient, see [Section 3.4](#).

### 4.9 Traversing object references

---

Once an object has been retrieved, i.e. loaded from the database through a class extent or a persistent static field, JB ensures that the object is in memory and its value is up-to-date with respect to the current transaction. Remember that this is true only if fields are accessed and updated through methods.

For example:

```
Person John;
...
System.out.println(John.getAge());
...
Person spouse = John.getSpouse();
transaction.begin();
John.setSpouse(Mary);
```

*// in a read-only transaction*  
*// John is loaded from the*  
*// database in a transparent way*  
*// John is already in the*  
*// database cache*  
*// John is reloaded in a transparent*  
*// way in order to ensure transaction*  
*// consistency*

One can directly access an object stored in the database from another object pointing to it. When loading an object from the database, that points to another object which has already been loaded into the JB cache, the object being loaded is simply set to point to the object in the cache through the corresponding reference variable. If, however, the object pointed to has not been loaded, a *shadow object* is created for it. The fields of a shadow object are not loaded until the object is explicitly accessed. Shadow objects are read in a transparent way.

### Retrieving array fields

When an object containing an array field is accessed, the array is treated as a reference field and a shadow array is created for it. When the array is effectively accessed in the program, it is loaded from the database in a transparent way to the programmer.

For an array of objects, all elements of the array are loaded as shadow objects from the database when the array itself is accessed. Array elements are loaded in a transparent way if their fields are accessed.

### Updating persistent objects

A persistent object is updated as usual through private methods of the class, like any transient object. Once a persistent object has been modified, it is updated in the database at commit or validate time. If an abort is done, the database is kept unchanged.

Example :



---

## Deleting persistent objects

---

```
Person Adam;
...
transaction.begin();
Adam.setName("Adam Smith"); // the object is implicitly loaded
                             // from the database, and updated.
Adam.getSpouse().setName("Eve Smith");
transaction.commit();        // The database is changed now.
```

### 4.10 Deleting persistent objects

---

A persistent object can be deleted from the database with the `delete` method of the class `Database`. Only persistent objects can be deleted. An attempt to delete a transient object raises an exception.

Example :

```
Person John; // Assuming John refers to a persistent object
...
transaction.begin();
...
Database.delete(John); // John is read from the database.
                       // John will be deleted from the database.
                       // John is kept in memory as a transient object.
Database.persist(John); // A new persistent object is created
transaction.commit();   // The database is changed now.
```

For the JB/Native and JB/JDBC versions, a persistent object cannot be deleted from the database as long as a stored object references it. An attempt to delete a persistent object referred to by another persistent object will raise the exception `DeleteReferencedObjectException`.

In order to delete a referenced persistent object from the database, the application program must first remove all references to such an object.

Since deletion is carried out immediately, as opposed to writing objects that are deferred until a commit, the references to the object to be deleted must be removed immediately. One way to do this is to set any

reference immediately to `nil` with the method `storeObject` of the class `Database`, see [Section 5.1](#).

For example, suppose objects `paul` and `mary` point to each other through the attribute `spouse`. To delete `mary` from the database, write the following :

```
...                // connection is established, base is opened
                   // and transaction begins;
Person paul,mary;
...                // paul and mary are loaded from the
                   // database;
paul.setSpouse(null); // reference to mary is removed;
database.storeObject(paul); // paul is immediately updated;
Database.delete(mary); // mary can be deleted, assuming the only
                       // reference to it has been removed;
...
transaction.commit(); // Everything is made persistent in
                       // the database;
```

Deleted objects remain in memory as `transient` objects.

With JB/O2, any persistent object may be deleted. Deletion is performed immediately as opposed to writing objects. If references to a deleted object still remain in the database, they behave as `nil` references.

### 4.11 Locking objects

---

JB uses three types of locks on objects: *shared locks*, *update locks* and *exclusive locks*. No locks are acquired on objects in a read-only mode.

Different concurrent transactions can acquire shared locks on the same object. Shared locks are automatically acquired when objects are loaded from the database inside a transaction. They are released when the transaction terminates (`abort`, `commit` or `validate`).

A transaction must request an exclusive lock on an object in order to be able to update it. In JB, such locks are acquired at commit time, when the objects are actually written from the cache to the database.

---

## Locking objects

---

An update lock can be acquired explicitly before commit time through the method `lock(Object)` of class `Transaction`.

In order to assure that an exclusive lock can be acquired at commit time for updated objects, and avoid deadlocks, update locks for such objects should be acquired by the application program as soon as possible.

---

### Note

---

Update locks exist only for JB/Native and JB/JDBC versions, and are database dependent.

---

### Avoiding deadlocks

The method `lock` acquires an update lock for an object in a single database operation (for JB/O2, an exclusive lock is acquired while for JB/JDBC and JB/Native an SQL `select for update` is done). This method is used to prevent deadlocks from occurring.

For example, if two transactions **T1** and **T2** access the same object `john`, and `john` is updated in **T1** and **T2**, at commit time, a deadlock will occur, since **T1** will wait for **T2** to release its share lock on `john` and **T2** will wait for **T1** to release its update lock on the same object `john`. This situation is illustrated in Figure 4.2. To avoid such a deadlock, you must lock the object before accessing it.

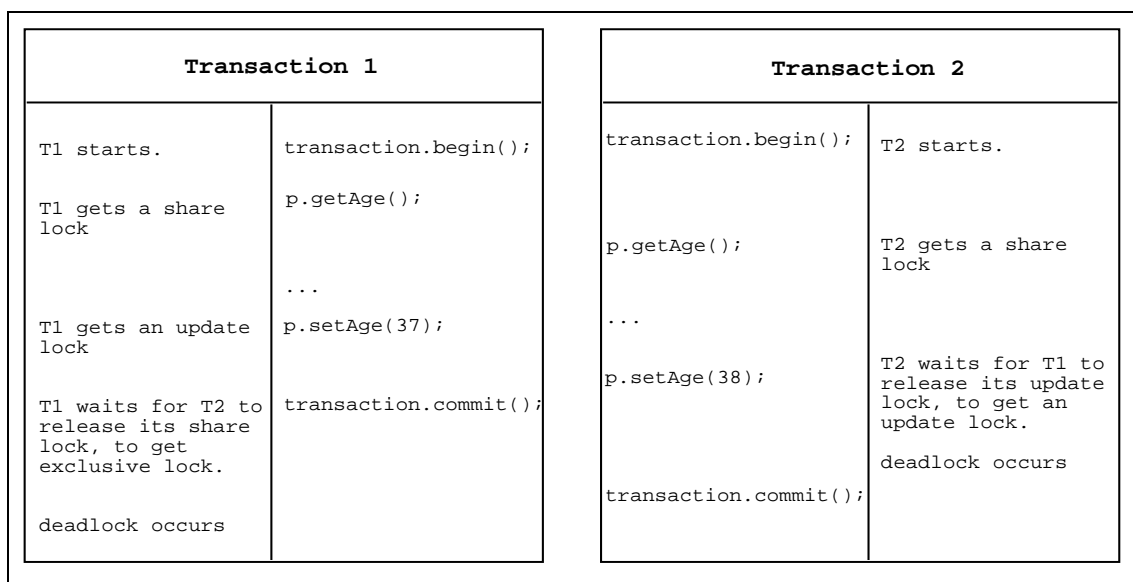


Figure 4.3: Deadlocks

In order to prevent deadlocks in a multi-user application, before attempting to update an object, an application should acquire a lock for the object.

Example :

```
Person p;
...
transaction.begin()
transaction.lock(p); // The object is loaded from the database,
                    // flagged as updated in the cache and
                    // and locked in the database.

p.getAge();
p.setAge(38);
transaction.commit(); // An exclusive lock on p is requested. If
                    // shared locks are held on it, the
                    // application waits until they are released.
                    // P is then finally written from the cache to
                    // the database and all locks are released
                    // at the end of commit.
```

An attempt to lock an object in a read-only transaction will raise the exception `NoActiveTransactionException`.

### Locking extents

Objects retrieved from a class extent inside a transaction are accessed through the method `nextElement()` applied to the extent enumeration. To allow objects loaded through an extent to be locked as they are accessed, and therefore avoid deadlocks, the method `lock` is provided in the class `Extent`.

Example :

```
transaction.begin();
Person p;
Extent personExtent = Extent.all("Person").\
where("this.name = \"John\" and this.age = 36");
personExtent.lock();
p = (Person)e.element();
p.setAge(37);
transaction.commit();
}
```

---

## Activate and prepareToWrite

---

### 4.12 Activate and prepareToWrite

---

Two methods may be written by a programmer for any persistent capable class which takes control when a persistent object is loaded to memory and before it is written in the database.

#### **activate**

The method `activate` is automatically called when an object is loaded from the database into the memory cache.

It allows the application to initialize transient fields.

#### **prepareToWrite**

The method `prepareToWrite`, as its name indicates, is automatically called just before the object is written to the database. It allows the user to define finalization routines.

### 4.13 Object identifiers

---

References to objects may be passed between various applications. JB provides a simple way to uniquely identify an object, together with operations to return an object's identifier (oid) and fetch an object using its oid.

Two methods are defined in the class `Database` for that purpose :

These methods are:

```
public Oid localOid(Object);  
public GlobalOid globalOid(Object);
```

The first method returns a local identifier that uniquely identifies an object in a database. The second method returns the global object identifier, that can be used across different databases. The global identifier contains information about the system (server and base) where the corresponding object is stored. An additional method is provided in the `Database` class to retrieve an object given its local identifier:

```
public Object retrieveObject(Oid oid);
```

For example:

```
GlobalOid oid;
...
database.connect();
database.open(oid.getBase());
...
Person john = database.retrieveObject(oid);
```

---

### Note

---

These methods are not supported by the JB/O2 version.

---

## 4.14 JDBC driver facilities

---

The JB/JDBC version depends on the particular JDBC drivers being used. Not all JDBC drivers provide the same features. To help you write generic applications with the JB API, we provide methods that ascertain which JB features exist for a particular JDBC driver.

This information is retrieved with the `getProperty()` method of the `Database` class. An object of the `JRBProperty` is returned.

The methods below can be called on this object:

`hasTransaction()` : returns true if the transaction feature is supported.

`hasTransactionIsolation3()` : returns true if the transaction isolation level 3 is supported.

`hasPrivileges()` : returns true if the user permission is supported.

---

## Exceptions

---

**hasExtentOfInterface()** : returns true if the user can use extent on interface.

**hasJRBlocks()** : returns true if the JB locking properties is supported.

**hasAllExceptions()** : returns true if all the following exceptions are supported.

- **hasDeleteReferencedObjectException()** : returns true if this exception is supported.

- **hasPermissionException()** : returns true if this exception is supported.

- **hasDeadLockExceptions()** : returns true if this exception is supported.

If one of the above features is not supported, the operation in the underlying database is a **noop**.

### 4.15 Exceptions

---

There are basically two kinds of exception, those raised by the JB runtime system itself and those raised by the underlying database system that are caught by the runtime and propagated to the application program after having been encapsulated into a DBMS exception.

Below, we list the exceptions that can be raised by the runtime system. All these exception classes inherit the class **DBRuntimeException**.

**ActiveTransactionException** : thrown when the user attempts to start a new transaction and there is a transaction already active.

**DBMSException** : thrown when an error or an exception is raised by the underlying database system. The message error returned by the underlying database server is reported when this kind of exception is raised.

**DatabaseAlreadyOpenException** : thrown when the user attempts to open a database while there is already a database opened.

**DeadLockException** : thrown when a deadlock occurs in the underlying database server.

**DeleteReferencedObjectException** : thrown when one attempts to delete a referred object.

**InvalidCallException** : thrown when a JB API call is made in a bad context (for example, you try to connect to another database when already connected).

**InvalidCallbackUsageException** : thrown when a call to JB API is made during execution of `prepareToWrite` or `activate` methods.

**InvalidExtentPredicateException** : thrown when a syntax error is discovered in a `where` clause of an extent (OQL or SQL syntax error)

**NoActiveTransactionException** : thrown when the user attempts to perform an update operation inside a read-only transaction.

**NoClassMetaDataException** : thrown when you manipulate an object of an imported class, and when you have forgotten to patch the class before recompiling it.

**NoManagedExtentException** : thrown when you access the extent of a class imported without extent (only in JB/O2)

**NotImportedClassException** : thrown when the user attempts to perform a persistent operation on a class that was not imported.

**NotSingletonExtentException** : thrown when the user attempts to retrieve the unique element of a non singleton extent.

**ObjectFaultException** : thrown when the user attempts to access an object that has been deleted inside the same transaction.

**PermissionException** : thrown when the user attempts to perform an operation he/she is not authorized to perform.

**ShadowObjectException** : thrown when one attempts to write a shadow object.

**StringOutOfRangeException** : thrown when one attempts to store a string which is larger than the system or user defined limit size.

**TransactionAbortedException** : thrown when the current transaction has been aborted due to a fatal error in the underlying database.

**TransientObjectException** : thrown when one attempts to access, lock or delete a transient object.



---

## User Administration

---

The following exceptions are thrown in very special case:

**DataDomainException** : thrown when data are retrieved from database and data type length is too large to fit in the Java data type. This case can occur only when data in the underlying database has been modified by another non JB program (for example an O<sub>2</sub> database is modified by a Java and a C++ application).

**DBUncheckedException** : thrown when an internal error is raised by the JB runtime. Normally, this error must be not thrown. In this case, call the support line.

### 4.16 User Administration

---

For the JB/NATIVE version, the JB API offers administration facilities to add and update user information. The class `User` in the `com.ardentsoftware.jb.api.admin` package provides the methods `addUser`, `addLogin`, `changeLogin` and `dropUser`. They allow new users and their corresponding passwords to be added and updated in the database, as illustrated below.

```
database.open("base1");  
...  
User.addUser("john");  
User.addLogin("john","john");  
...
```

Only a super user can add new users to the database. For the complete documentation of user administration commands, refer to the HTML documents.

The user administration facilities are not supported by JB/JDBC and JB/O<sub>2</sub>.

A call to user administration operations for JB/JDBC and JB/O<sub>2</sub> is a noop.

### 4.17 User Access Administration

---

JB/NATIVE offer an API to manage user access rights. The JB security model follows the SQL-92 security model. There are six kinds of operations that can be granted or revoked from a user:

**Import:** the user is enabled/disabled to import java classes.

**Access:** the user is enabled/disabled to read persistent objects.

**Update:** the user is enabled/disabled to write persistent objects.

**Delete:** the user is enabled/disabled to delete persistent objects.

**All:** the user is enabled/disabled to read, write and delete persistent objects.

**Grant:** the user is enabled/disabled to perform all the previous operations and to give grant permission.

These operations are performed through the `Permission` class of the `com.ardentsoftware.jb.api.admin` package.

The `jb importer` of a database is the user that creates and initializes the database and imports classes and has all the rights on imported classes. This user can grant or revoke access rights from other users on classes imported to the base.

A user must be connected to a database to be able to set access rights and all transactions must first be committed or aborted, otherwise an exception is raised.

At runtime, if a user tries to execute a non authorized operation, an exception `PermissionException` is raised.

The example below grants access rights to the user `john` for the class `Person`.

```
Permission.enable(Permission.ACCESS, "john", "Person");
```

Access rights on all imported classes can be granted or revoked through a shortcut as follows.

---

## User Access Administration

---

```
Permission.enable(Permission.ACCESS,"john",Permission.ALLCLASSES);
```

Other methods allow you to inspect which access rights a user has for a given class, as illustrated below.

```
if (Permission.can((Permission.ACCESS,"john","Person")) {  
    Person spouse = john.getSpouse();  
    ...}
```

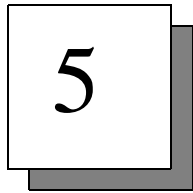
JDBC drivers do not support privilege SQL statements. In order to test if user permission is valid, use the `JRBProperty` class (see [Section 4.14](#)).

The JB/O<sub>2</sub> and JB/JDBC versions do not support user access permission administration. In this case, calling user permissions does nothing.

For the complete `Permission` class documentation refer to the HTML documentation.



---



# Advanced Programming

---

This chapter presents advanced techniques for managing persistence in a non transparent way and how to optimize your application in terms of space and time. It describes how to use JB not only for storing Java objects, but also Java bytecode, and how to access the JB catalog, which contains information about Java and Database schemas.

This chapter is composed of the following section:

- *Persistence in a non transparent way*
- *Tuning Memory Management*
- *Storing bytecodes in a database*
- *Using O2 schemas with Java applications*
- *Inspecting the system catalog*

## 5.1 Persistence in a non transparent way

---

You can choose to manage retrieval and update of persistent objects in a non transparent way. You may have different reasons to not use the transparent API, for example:

- You do not want to encapsulate fields of imported classes.
- You want to directly control memory management.
- You want to generate JB code.
- You need very high performance and optimize your code by hand.

For such cases, you should use the `o2jb_import` tool with the option `-noppp`, which turns off the post processing of the import tool.

Storage and retrieval of Java objects from the database in a non transparent way is achieved through the methods `access`, `persist` or `markModify` which are declared in the class `Database`.

In the same program you can mix transparent and non-transparent API. When performance is an issue, some explicit operations described in this chapter may lead to more efficient code.

An object can become persistent only if it is an instance of an imported class. Objects can be made persistent, loaded, modified and deleted from the database.

An object in a JB application can be in one of the following states : `transient`, `persistent` or `shadow` (see [Section 4.4](#) for a description of the first two states).

A shadow object is an object that has been partially loaded from the database when a persistent object pointing to it was loaded. It corresponds to a persistent object in the database and has a corresponding handle in the JB cache. Fields of a shadow object are not loaded from the database. A shadow object is loaded in a non transparent way if the `access` method is called.

The method `isShadow(Object)` of class `Database` is provided to allow the application program to query if an object is a shadow object. This method returns `true` if the object is a shadow object, `false` otherwise.

---

## Persistence in a non transparent way

---

### Accessing a persistent object

The method `access` of the class `Database` must be called once before accessing a persistent object. This method loads the object to memory or does nothing if the object is already in memory.

### Updating a persistent object

The method `markModify` of the class `Database` must be called to indicate that a persistent object is updated. Only mark modified objects are rewritten into the database when the transaction commits.

The example below shows different state transitions that are detailed in subsection [State transition of database events](#).

```
transaction = new Transaction();
...           // p is accessed through another object
              // and is a shadow object.
Database.access(p); // p is accessed in read-only
                  // transaction. Its fields can be
                  // accessed.
Database.access(p); //p is not reloaded, because it is
                  // already in memory.
...
transaction.begin(); //Start a transaction
...                 // p becomes a shadow object, because it
                  // might be updated by another
                  // transaction.
```

```
Database.access(p);           //p is reloaded
Database.markModify(p);       //p is locked in the database and will
                               // be written at commit time.
if(...)
    transaction.commit();     //p is written in the database.
else
    transaction.abort();      //rollbacks the modifications in the
                               //database.
                               //p is no more in the JB cache.
...                           // Read-only transaction
transaction.begin();         //start a new transaction
...                           //Access p through the class extent
Database.markModify(p);
transaction.validate();      //p is still in the JB cache
...
Database.access(p);         //p is not reloaded.
...
transaction.begin();        //Start a new transaction
                               // p is still in the JB cache, but as
                               // a shadow object.
Database.access(p);         //Reloading p from database ensures
...                           //consistency.
transaction.validate();
```

When a transaction starts, all persistent objects in your cache become shadow objects. Objects loaded in a read-only transaction must be reloaded from the database inside a transaction before being updated.

### Persistent arrays

When a large array containing references to objects is loaded from the database using method `access` of class `Database`, a large amount of memory space and time is consumed in creating shadow objects for each array element and in loading each element from the database. The same applies for an array of primitive type elements, although to a lesser extent, since it does not involve the creation of shadow objects.

If only a few array elements are actually needed by the application, you can improve performance by using the static method `access`, with the index of the array, defined in class `Database`, which allow array



---

## Persistence in a non transparent way

---

elements to be accessed on demand. This avoids loading all the array elements as shadow objects.

Consider the following example:

```
class Student extends Person {
    Professor responsible;
    int mark[];
    Person friends[];
}
```

When an instance of `Student` is accessed, e.g `Bob`, field `friends` is loaded as a shadow array but the array elements are not accessed. A shadow array has the same size as the array in the database, but its elements are not loaded as shadow objects.

```
Database.access(Bob);
```

Figure 5.1 presents the persistent object `Bob`, an instance of class `Student`, after it was read from the database into memory.

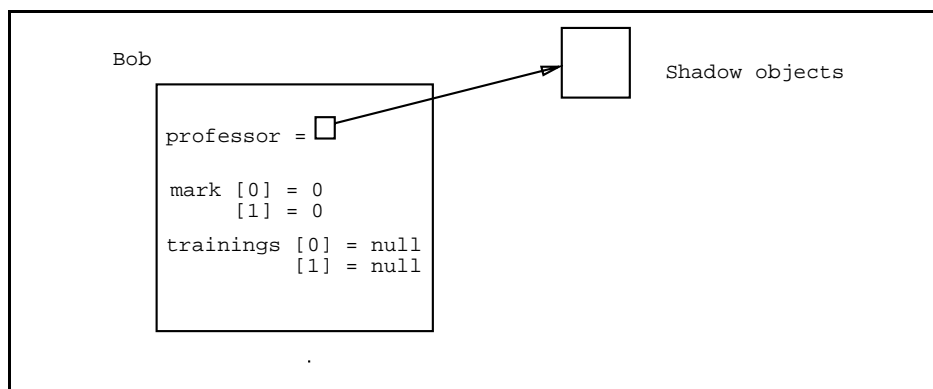


Figure 5.1 : Persistent object `Bob` in memory

You can also access a particular element of an array without reading the entire array, by calling the method `Database.access(array, index)`. The example below shows how array elements can be retrieved on demand.

```
Student Bob
Extent studentExtent;
...
studentExtent = Extent.proper("Student").
                    where("this.name = \"Bob\"");
if (studentExtent.size() == 1) {
    Bob = (Student)studentExtent.element();
}
int nbFriends = Bob.friends.length; // Retrieves the number
                                     // of elements in the array

Person John;
Database.access(Bob.friends,1);
John = Bob.friends[1]; // The reference to John is loaded
                       // from the database. John is a
                       // shadow object. Only one object
                       // is created in memory.
Database.access(John); // John is loaded.
```

Similarly, when an array is written into the database, it may take a long time to write all the entries of the array. You can use the `persist` or `markModify` methods, with the index parameter, of the `Database` class to write only a few elements of the array into the database.

The example below shows how some elements of the array can be written.

```
Student Bob;
... // John and Mary are persistent objects
    // and are loaded in the cache.

transaction.begin();
Database.markModify(Bob);
Bob.friends = new Person[10];
Bob.friends[0] = John;
Bob.friends[3] = Mary;
Database.persist(Bob.friends,0);
                                     // update the entry 0 of the array
Database.persist(Bob.friends,3);
                                     // update the entry 3 of the array
transaction.commit(); // Only two entries are written into
                       // the database the reference to
                       // John and to Mary.
```

---

## Persistence in a non transparent way

---

### String fields

This section only applies to JB for relational databases.

A Java string field may be stored in the database as either `varchar` or `longvarchar` depending on the size of the string. By default, string fields are stored in a table columns of type `varchar`. This method greatly speeds up retrieval and storage of string fields.

A column of type `varchar` can hold strings of character size no greater than 255 for Sybase and 2000 for Oracle. An attempt to store a larger string will raise an exception `StringOutOfRangeException`. The default type can be redefined by the user as `long varchar`, so that larger strings (with up to  $2^{31}-1$  characters) can be stored. This redefinition should be performed if a given string field may exceed the default limit.

The string field type may be redefined in a configuration file of the import tool. Syntax for redefining string types is given in [Section 3.4](#).

### State transition of database events

Using JB in a non transparent way requires the user to implement manually the the persistence of objects. To that end, it is essential to know the state of each object and which JB-API to apply at the appropriate time. Figure 5.2 presents a state transition diagram for an object. A circle represents the object in a given state. An arrow corresponds to an event (`access`, `persist`, `markModify`, `lock`, `delete`) and points to the next state of the object after the event.

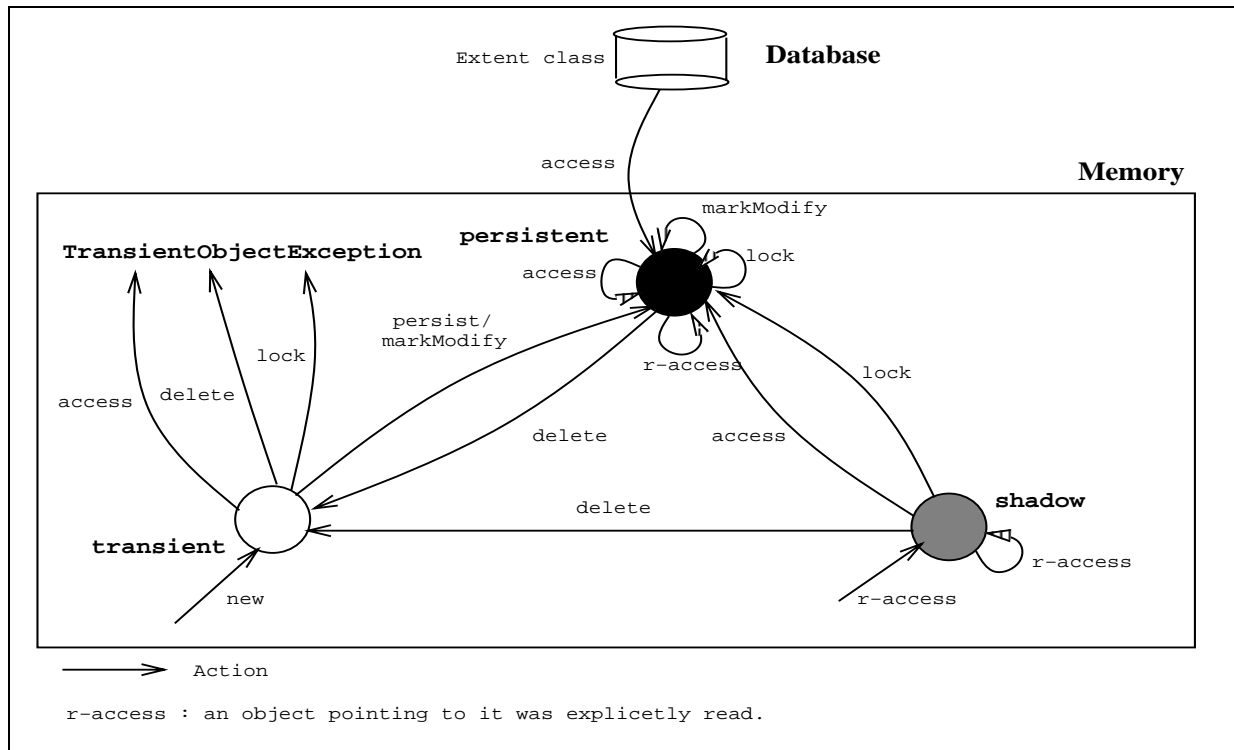


Figure 5.2 : State transition of database events

Figure 5.3 on the following page illustrates state transitions of events corresponding to transactional operations (**begin**, **abort**, **commit**, **validate**).

---

## Tuning Memory Management

---

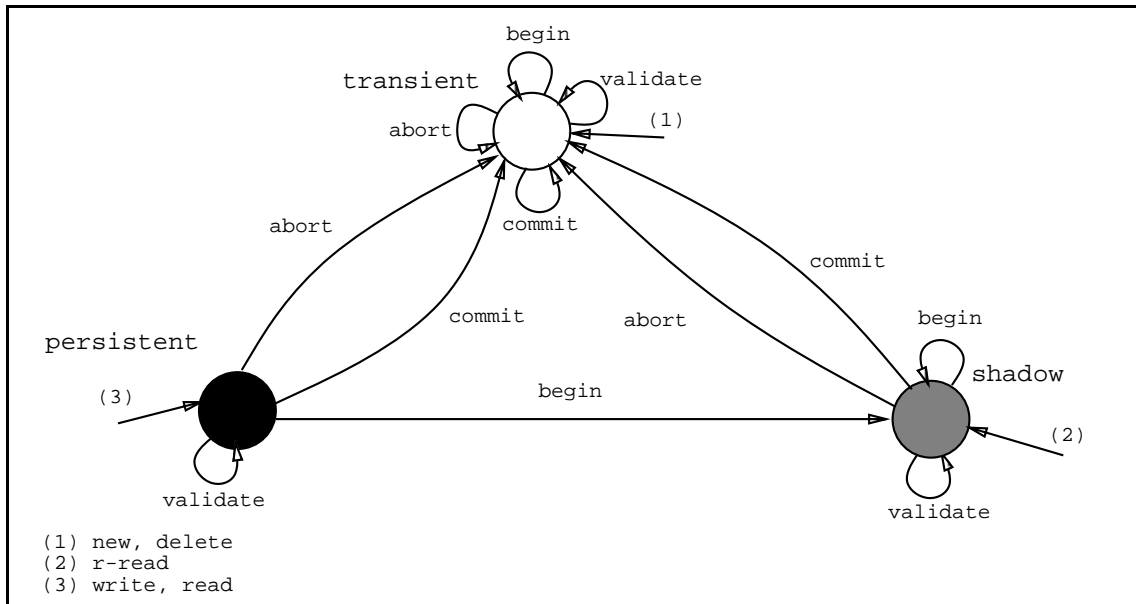


Figure 5.3 : State transitions of transactional events

## 5.2 Tuning Memory Management

---

### Releasing objects from memory

Objects accessed or updated are kept in the object cache. The object cache is freed at commit and abort time, whereas the `validate` method does not free the cache.

If you want to optimize memory space usage, you can free the object cache on demand. When an object is no longer needed by the current database transaction, the programmer can remove the object from the object cache. This object can then be garbage collected.

To remove an object from the object cache, the `release` method of class `Database` must be explicitly called on the object. Before removing the object from the object cache, it is written to the database if its state is `persist` or `markModify`.

After `release`, the object becomes `transient`. A subsequent `persist` creates a new object in the database at commit time. Reading a released, i.e. `transient`, object raises the exception

**TransientObjectException.** Released objects are not deleted from the database, but from the object cache only. They can be retrieved from the database by using one of the data retrieval facilities available. The example below illustrates the removal of objects from the object cache.

```
Person p;
...
Database.access(p);
    // The object is accessed in the database and is persistent.
Database.release(p);
    // p becomes transient and is removed
    // from the object cache.
Database.persist(p); // A new persistent object is created
    // and kept in the cache.
Database.release(p);
    // p is written to the database before being
    // removed from the cache.
Database.access(p); // An exception is raised, only shadow and
    // persistent objects can be loaded.
```

### Writing objects to the database

As described above, updating objects in the database is performed through the method `markModify` of class `Database`. The actual modifications to persistent objects are written in the underlying database only at commit or validate time.

For transactions modifying a large number of objects in the database, the commit or validate operation may take a long time and force the application to wait before being able to resume its work.

To avoid long waits, you can utilize incremental data transfers to the database, in order to reduce the amount of data to be transferred at commit or validate time.

If you want to write an object to the database immediately, instead of deferring the write to the commit, the method `storeObject` of class `Database` can be used after calling the method `markModify` with the object as an argument. This object is immediately written to the database rather than at commit or validate time. Of course, if the transaction

---

## Storing bytecodes in a database

---

aborts, the corresponding database transaction is aborted and modifications to the underlying database are rolled back.

The method `storeObject` can thus be used to immediately transferred to the database objects that are *ready* to be stored, i.e. that are not expected to be further modified. Together with the `release` method, this allows the programmer to distribute and optimize resources in space and time for a given application. The method `storeObject` does not release the object cache.

Another important use of method `storeObject` is related to the deletion of referenced objects, as illustrated in [Section 4.10](#).

### 5.3 Storing bytecodes in a database

---

It is possible to store Java bytecode in a database. Stored bytecode can be loaded into the runtime by means of the class `DatabaseClassLoader`. This class extends the standard Java class `ClassLoader` and provides a mechanism to load bytecode and pass it to the Java runtime, thereby making it possible for the program to manipulate objects belonging to the loaded class. The user may explicitly use the class `DatabaseClassLoader` to load a particular class. It is useful to load stored bytecode, when the database contains objects of a class which is not available to the program from the CLASSPATH. If the program accesses such an object, the `DatabaseClassLoader` loads the implementation of the class from the database, allowing the object to be properly used.

The methods and constructors of classes loaded by a `DatabaseClassLoader` may reference other classes. To determine the classes referred to, the Java Virtual Machine calls the method `loadClass` of the `DatabaseClassLoader` that originally loaded the class.

Lets provide an example of this feature. Suppose we have a class `C1` which implements the interface `DatabaseRunnable` and defines a method `main(...)` which stores and retrieves objects of type `C2`. Classes `C1` and `C2` are stored as bytecode in the database. The interface `DatabaseRunnable` defines a method `runFromDatabase(...)` and the implementation of this method in class `C1` performs a call to the method `main(...)` of class `C1`. To use classes `C1` and `C2`, an external program only needs to access the `DatabaseRunnable` interface; it creates an object of type `C1` and calls

its `runFromDatabase(...)` method and the proper action is performed. The essential steps are shown below.

```
//      constructor for a database class loader
DatabaseClassLoader loader =
    new DatabaseClassLoader(database);
//      obj will become of class C1
Object obj = loader.loadClass("C1").newInstance() ;
//      dr will allow running obj from an external program
DatabaseRunnable dr = (DatabaseRunnable)obj ;
//      run the main of obj
obj.runFromDatabase(...) ;
```

When the program tries to access an object of type `C2`, the class `C2` is loaded by the same `DatabaseClassLoader` which was used to load class `C1`.

The `DatabaseClassLoader` object is valid as long as the connection to the database is established. Therefore all the classes you want to load with the `DatabaseClassLoader` must be loaded during the same connection.

Below we provide an complete example including the interface `DatabaseRunnable` and two classes `C1` and `C2` as well as an external program.

```
interface DatabaseRunnable {
    void runFromDatabase (Database db,
                          String myBase);
}
```



---

## Storing bytecodes in a database

---

```
//          class C1
import java.util.*;
import com.ardentsoftware.jb.api.*;
//          implements the interface DatabaseRunnable
public class Prog1 implements DatabaseRunnable {
//          and contains a method main
    public static void main (Database database,
        String myBase) throws DBException {

        Transaction transaction = new Transaction();
//          constructor for object of class C2
//          will automatically load class C2
        Rectangle rect = new Rectangle(10,14);
        Database.persist(rect);

        transaction.commit();

        database.close();
        database.disconnect();
    }

//          method runFromDatabase of interface
//          DatabaseRunnable which calls main
    public void runFromDatabase(Database database,
        String myBase) throws DBException{
        main(database, myBase);
    }
}
```

```
//          class C2
import java.util.*;
import com.ardentsoftware.jb.api.*;

public class Rectangle {
    int height;
    int width;
    public Rectangle (int h, int w) {
        height = h;
        width = w;
    }
}
```

```
//          external program
import com.ardentsoftware.jb.api.*;

class MainLoad {
    public static void main (String args[]) throws
    DBException {

        Database database = new
            Database(Database.O2,mySystem,"", "");

        database.connect(); // Connection to the server
        database.open(myBase);

        try {
            System.out.println("Calling loadClass ... ");
//          DatabaseClassLoader for class C1
            DatabaseClassLoader loader = new
                DatabaseClassLoader(null);
            Class loaded = loader.loadClass("Prog1",true);
//          object x is of class C1
            Object x = loaded.newInstance();
            DatabaseRunnable dr = (DatabaseRunnable) x;
//          calling method in C1 to execute main
            dr.runFromDatabase(database, myBase);
            System.out.println("DONE!");
        }
        catch(ClassNotFoundException e) {
            e.printStackTrace();
        }
        catch(IllegalAccessException ee) {
            ee.printStackTrace();
        }
        catch(InstantiationException eee) {
            eee.printStackTrace();
        }
        database.close();
        database.disconnect();
    }
}
```

All the referenced classes are loaded by the same Class Loader as the originally loaded class. The `DatabaseClassLoader` can load classes either from a file system or from a database but once a class has been

---

## Storing bytecodes in a database

---

loaded using the Java Virtual Machine Class Loader, all the classes it refers to must exist in the file system.

To store bytecode in a database use the `o2jb_store_bytecode` tool. You can store bytecode of specific classes or of an entire package. For example, to store in database `java_base` the bytecode of classes `Person` and `Student` defined in package `example`, the following code may be used:

```
o2jb_store_bytecode -url o2:myserver:mssystem -base java_base
                    -schema java_schema example.Person example.Student
```

Alternatively, the bytecode of all the `example` classes can be stored with option `-package example`.

The bytecode of one or more classes can be deleted from the database as illustrated below:

```
o2jb_delete_bytecode -url o2:myserver:mssystem -base java_base
                    -schema java_schema -package example
```

Contrary to `import` and `unimport`, bytecode storage and deletion is not applied recursively to the superclasses and subclasses of input classes.

The bytecode of a given class can be stored even if that class has not been imported. Conversely, a class can be imported into a database even if its bytecode is not stored in the same database.

## 5.4 Using O<sub>2</sub> schemas with Java applications

---

With JB/O<sub>2</sub> you can access any O<sub>2</sub> schema with a Java application. This includes schemas generated through C++ applications and with O<sub>2</sub>C.

In order to use an O<sub>2</sub> schema with a Java application, you must install the O<sub>2</sub>Java schema in your system. Use the `o2dba_schema_load` tool (see the *O<sub>2</sub> System Administration Reference Manual*) to load the schemas from the `$O2HOME/o2schemas` directory :

```
o2dba_schema_load -f -sources -file $(O2HOME)/o2schemas/o2java.dump
-system O2SYSTEM -server O2SERVER
```

The tool `o2dba_schema_load` asks you for the name of the volume where it will install the schema. You can use the volume `CatalVo1` for this purpose by default.

---

### **Warning !**

---

To use `o2dba_schema_load`, you must launch the O<sub>2</sub> server in single user mode.

---

To execute a JB/O<sub>2</sub> application on an O<sub>2</sub> schema that was not created by the `o2jb_create_schema` tool, one must first load the O<sub>2</sub> supplied file, `o2java.dump` as described above and then import the classes of this predefined schema `o2java` into your O<sub>2</sub> schema. To do this launch `o2shell` in your O<sub>2</sub> system and execute the command:

```
import schema o2java class PersistentString, PersistentHashtable,
PersistentVector, o2_list_Object
```

You can now use your schema with a Java application.

---

## Inspecting the system catalog

---

### 5.5 Inspecting the system catalog

---

#### Inspecting imported classes

Once a class has been imported into the database, you can retrieve information on the imported class using the tool `o2jb_dump_classes`. One use of this tool is to compare the new version of a class that has evolved with the version imported to the database. This allows programs to check for compatibility between the imported version and the Java version. Imported classes can be inspected using the `o2jb_dump_classes` tool. The structures of the imported classes are dumped as they were defined when the class was imported, taking into account any renaming described in a configuration file.

Consider the following example, for the classes `Person` and `Student`:

```
class Person {
    int age;
    String name;
    Person spouse;
    Person children[];
    transient int agegroup;
}
class Student extends Person {
    Person professor;
}
```

Using the dump classes command:

```
o2jb_dump_classes -url o2:mysqlserver:mssystem -schema schema1
```

The following is generated in the output file `schema1.classes`:

```
package schema1;
public class Person
{
    private int age;
    private String name;
    private Person spouse;
    private Person Children[];
}
public class Student extends Person {
    private Person professor;
}
```

Using the command with Sybase:

```
o2jb_dump_classes -url sybase:java_store -base base1
                  -user john -passwd john_passwd
```

The following is generated in the output file `base1.classes`:

```
class Person {
    int age;
    String name;
    Person spouse;
    Person children[];
}
class Student extends Person {
    Person professor;
}
```

For complete details of the `o2jb_dump_classes` tool see Chapter 6, Section [o2jb\\_dump\\_classes](#).

### Inspecting database structures

The structures in the database, i.e. O<sub>2</sub> classes or tables generated for storing instances of imported classes can be inspected using the `o2jb_dump_schema` tool.

---

## Inspecting the system catalog

---

For the following `Person` and `Student` classes:

```
class Person {
    int age;
    String name;
    Person spouse;
    Person children[];
    transient int agegroup;
}
class Student extends Person {
    Person professor;
}
```

Using the `dump schema` command:

```
o2jb_dump_schema -url o2:myserver:mssystem -schema schema1
```

This command generates in the output file `schema1.oql` :

```
class Person inherit Object
public type tuple(
    age: integer,
    name: string,
    spouse: Person,
    Children: o2_list_Person);

class Student inherit Person
public type tuple(
    professor: Person)
```

With this command for Sybase :

```
o2jb_dump_schema -url sybase:java_store -base base1
                -user john -passwd john_passwd
```

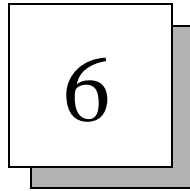
The following is generated in the output file `base1.sql`:

```
create table Person (  
    oid OID,  
    primary key (oid),  
    class CLASS,  
    age int,  
    name STRING,  
    spouse_oid REF,  
    spouse_class CLASS,  
    foreign key (spouse_oid) references Person,  
    children_oid ARRAY,  
    children_flag FLAG,  
    children_size int,  
    foreign key (children_oid) references Person_A  
)  
create table Student (  
    oid OID,  
    primary key (oid),  
    class CLASS,  
    professor_oid REF,  
    professor_class CLASS,  
    foreign key (professor_oid) references Person  
)  
create table Person_A (  
    oid OID,  
    primary key (oid),  
    size INT  
)  
create table Person_A_E (  
    oid REF,  
    foreign key (oid) references Person_A,  
    position INT,  
    element_oid REF,  
    element_class CLASS,  
    foreign key (element_oid) references Person  
)
```

For further details of the `o2jb_dump_schema` tool see Chapter 6, Section [o2jb\\_dump\\_schema](#).



---



## JB Tools Reference

---

This chapter details the usage of the JB tools. For each tool, we provide its syntax, the list of mandatory and optional arguments, a description of the tool and the environment variables and files related to the tool, if any.

---

### **Warning !**

For the JB/O2 version, with a Sparc Solaris platform, you must use JVM with native threads (available with Solaris 2.5 and higher). Set the environment variable `THREADS_FLAG` to `native`.

---

The tools are:

[o2jb\\_create\\_base](#)

[o2jb\\_create\\_importer](#)

[o2jb\\_create\\_schema](#)

[o2jb\\_delete\\_base](#)

[o2jb\\_delete\\_bytecode](#)

[o2jb\\_delete\\_importer](#)

[o2jb\\_delete\\_schema](#)

[o2jb\\_dump\\_bytecode](#)

[o2jb\\_dump\\_classes](#)

[o2jb\\_dump\\_schema](#)

[o2jb\\_export](#)

[o2jb\\_import](#)

[o2jb\\_init\\_base](#)

[o2jb\\_patch\\_class](#)

[o2jb\\_store\\_bytecode](#)

[o2jb\\_test\\_driver](#)

[o2jb\\_unimport](#)

---

## URL syntax

---

### URL syntax

---

**Summary** The `URL` is used for giving the necessary information for database server connection for all JB tools.

**Description** The `URL` is indicated after the `-url` options of all JB tools. The URL is a string of token separated by ":" character. The first token is the type of database system. Valid values are:

- **`o2`**
- **`jdbc`**
- **`oracle`**
- **`sybase`**

For `O2`, the second token is the name of the computer on which the `o2server` is running. The third token is the name of the `o2` system you want to connect to. For example:

```
o2:my_server:my_system
```

For JDBC, the syntax follows the recommendation of jdbc drivers `url`. For example:

```
jdbc:odbc:myDataSource
```

for the connection with the JDBC/ODBC bridge.

For JDBC and in all tools, you must also give the driver Java class name using option `-driverclass`. For example:

```
sun.jdbc.odbc.JdbcOdbcDriver
```

For Oracle, nothing else is necessary. The `url` is:

```
oracle
```

For Sybase, the second token is the name of the sybase system. For example:

```
sybase:java_store
```

## o2jb\_create\_base

---

**Summary** creates a new JB database. **Note** : The o2jb\_create\_base tool is not supported by the JB/JDBC version.

**Syntax** o2jb\_create\_base

```

    -user      user_name
    -passwd   user_password
    -base     base_name
    -size     base_size
    -url      url_string
    -schema   schema_name (For JB/O2 only)
    -device   device_name (For JB/SYBASE only)
    -log      log_name (For JB/SYBASE only)
    -logsize  log_size (For JB/SYBASE only)
    [-verbose]
    [-help]
  
```

### Mandatory arguments

**-user user\_name** : login name of an importer user created with the `create_importer` tool (see note on page 5.2).

**Note** : The **-user** option is ignored by the JB/O2 version.

**-passwd user\_passwd** : password of the importer user.

**Note** : The **-passwd** option is ignored by the JB/O2 version.

**-base base\_name** : name of the database to be created.

**-size base\_size** : amount of space allocated to the database in megabytes. **Note** : The **-size** option is ignored by the JB/O2 version.

**-url url\_string** : url permitting the connection to the underlying database. Only O<sub>2</sub>, Oracle and Sybase are supported by this command.

The following argument must be provided *for JB/O2 databases only*.

**-schema schema\_name** : name of the o2 schema of the created base.

The following arguments must be provided *for JB/SYBASE databases only*.

---

## **o2jb\_create\_base**

---

**-device device\_name** : logical name of the Sybase database device where the newly created database is to be put.

**-log log\_name** : logical name of the Sybase device that will be used to store database logs.

**-logsize log\_size** : amount of space allocated to the database log in megabytes.

### **Optional arguments**

**-verbose** : turns the verbose mode on. The import tool works by default in a silent mode.

**-help** : displays the usage of the tool.

### **Description**

The base creation tool creates a new JRB database in the underlying database system. After being created, a JRB database must be initialized with the `o2jb_init_base` tool.

The importer creating a database becomes the *owner* of the database. This is the only user authorized to import classes into the database.

A JB database is a logical concept that is implemented according to the type of the underlying database system:

JB/O2 : for O<sub>2</sub> database system, a JB database is a O<sub>2</sub> database.

JB/Oracle : for Oracle, a new tablespace is created as follows:

```
create tablespace base_name datafile 'base_name.dbf'  
size device_sizeM reuse;
```

```
create user base_name identified by user_passwd;
```

```
grant create session to base_name;
```

```
grant unlimited tablespace to base_name;
```

The user `base_name` is created to group all tables and associated objects (procedures, sequences, packages) that are created for imported classes into a separate JB database schema. This user takes the same password as that of the importer user creating the database. A separate tablespace is created per JB database so that different physical volumes are used for each JB database.

Different configurations can be defined using the appropriate administration tools and commands, provided the minimal configuration

defined by the commands above (relationship between the importer, the dummy user `base_name` and the tablespace `base_name`) is preserved.

For more information on tablespaces, schemas and database (re)dimensioning in general, refer to the *Oracle SQL Server Reference Manual*.

JB/Sybase : for Sybase, a new database is created as follows:

```
create database base_name on device_name =  
device_size log on log_name = log_size;
```

User `user_name` is the user that creates the database and becomes its owner. Redimensioning can be performed by the database owner through the appropriate commands and tools (e.g. the `alter database` command).

For more information on database devices and database (re)dimensioning in general, refer to the *Sybase SQL Server Reference Manual*.

**Files**           None.

**See also**       `o2jb_create_importer`, `o2jb_delete_base`, `o2jb_import`

---

## o2jb\_create\_importer

---

### o2jb\_create\_importer

---

**Summary** Adds a JB importer user having class import rights on the database he/she creates. **Note:**This functionality is not supported by the JB/JDBC and JB/O2 version.

**Syntax** `o2jb_create_importer`

```
-admin      admin_name
-adminpasswd admin_passwd
-user       user_name
-passwd     user_password
-url        url_string
[-verbose]
[-help]
```

#### Mandatory arguments

`-admin admin_name` :login name of a system administrator user of the underlying database.

**Note on the JB/Oracle version:** If you are connecting to a remote server, you must add the Oracle dblink to the user name, e.g.

```
o2jb_create_importer -admin sys@jrb -adminpasswd ora
... or set the Oracle environment variable TWO_TASK to your dblink.
The configuration of the dblink to use (in the example, jrb) is done
through Oracle's "SQL*Net Easy Configuration" utility in NT.
```

`-adminpasswd admin_passwd` : password of a system administrator user of the underlying database.

`-user user_name` :login name of the importer user to be created.

`-passwd user_password` : password of the importer user to be created.

**Note on the JB/Sybase version:** The user password must be greater than 6 characters and must not begin with a numeric.

`-url url_string` : url permitting the connection to the underlying database. Only Oracle and Sybase are supported by this command.

**Optional arguments**

`-verbose` : turns the verbose mode on. The import tool works by default in a silent mode.

`-help` : displays the usage of the tool.

**Description** The importer creation tool registers the new user and password in the database and grants him/her the appropriate rights to import classes and create new users in a database he/she owns. Many different import users can be created but an importer can import only to the database(s) that he/she creates.

**Files** None.

**See also** `o2jb_delete_importer`, `o2jb_create_base`,  
`o2jb_delete_base`,

`o2jb_import`, `o2jb_unimport`



---

## o2jb\_create\_schema

---

### o2jb\_create\_schema

---

**Summary**      Creates a new o2 schema.

**Syntax**        `o2jb_create_schema`

```
{ -url url_string |  
  -system O2 -server mysystem@myserver }  
-schema            schema_name  
[-verbose]  
[-help]
```

**Mandatory arguments**

-url *url\_string* : url permitting the connection to the o2 database.

-schema *schema\_name* : name of the o2 schema to create.

**Optional arguments**

-verbose : turns the verbose mode on. The create schema tool works by default in a silent mode.

-help : displays the usage of the tool.

**Description**    The schema creation tool creates a new o2 schema in the o2 database.  
**Note:** The o2jb\_create\_schema tool is only supported by the JB/O2 version.

**Files**            None.

**See also**        `o2jb_create_base`, `o2jb_delete_schema`

## **o2jb\_delete\_base**

---

**Summary** Deletes a JB database. **Note:** This tool is not supported with the JB/JDBC version.

**Syntax** `o2jb_delete_base`

```
-user      user_name
-passwd    user_password
-base      base_name
-url       url_string
-schema    schema_name (for JB/O2 only)
[-verbose]
[-help]
```

### **Mandatory arguments**

`-user user_name` : login name of the database owner (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

`-passwd user_passwd` : password of the database owner. **Note:** The `-passwd` option is ignored by the JB/O2 version.

`-base base_name` : name of the database to be deleted.

`-url url_string` : url permitting the connection to the underlying database.

The following argument must be provided *for JB/O2 databases only*.

`-schema schema_name` : name of the o2 schema of the created base.

### **Optional arguments**

`-verbose` : turns the verbose mode on. The import tool works by default in a silent mode.

`-help` : displays the usage of the tool.

**Description** The base deletion tool deletes an existing JB database. Data stored in the database are lost. Only the database owner is authorized to delete a given database. Only initialized databases can be deleted with the `o2jb_delete_base` tool.

**Files** None.

---

## **o2jb\_delete\_base**

---

**See also**    `o2jb_create_base`, `o2jb_init_base`

## o2jb\_delete\_bytecode

---

**Summary** Deletes the bytecode of a given class from a database.

**Syntax** o2jb\_delete\_bytecode

```

    -user          user_name
    -passwd        user_passwd
    -base          base_name
    -url           url_string
    -schema        schema_name (for JB/O2 only)
    -driverclass  driver_class_name (For JB/JDBC only)
    [-package     package_name]
    [-verbose]
    [-help]
    class_name*
  
```

### Mandatory arguments

**-user user\_name** : login name of an importer user created with the `create_importer` tool (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

**-passwd user\_passwd** : password of the importer user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

**-base base\_name** : name of the database from which the bytecode is to be deleted.

**class\_name\*** : list of qualified class names separated by a blank. These are the classes whose bytecode is to be deleted from the database.

**-url url\_string** : url permitting the connection to the underlying database.

**-schema schema\_name** : name of the o2 schema.

The following argument must be provided *for JB/JDBC only*

**-driverclass** : allows a driver Java class name to be specified.

### Optional arguments

**-package package\_name** : deletes the bytecode of all classes in the

---

## **o2jb\_delete\_bytecode**

---

package **package\_name** from the database. Different packages can be passed as arguments, each with a **-package** flag.

**-verbose** : turns the verbose mode on. The **import** tool works by default in a silent mode.

**-help** : displays the usage of the tool.

**Description** This tool deletes the stored bytecode of a class or a set of classes from the database. Only the database owner is authorized to delete bytecode from the database.

**Files** None.

**See also** **o2jb\_store\_bytecode**, **o2jb\_dump\_bytecode**

## **o2jb\_delete\_importer**

---

**Summary** Deletes a JB importer user. **Note:**This functionality is not supported by the JB/JDBC and JB/O2 version.

**Syntax** `o2jb_delete_importer`

```
-admin      admin_name
-adminpasswd admin_passwd
-user       user_name
-url        url_string
[-verbose]
[-help]
```

### **Mandatory arguments**

`-admin admin_name` : login name of a system administrator user of the underlying database (see note on page 5.2 ).

`-adminpasswd admin_passwd` : password of a system administrator user of the underlying database.

`-user user_name` : login name of the importer user to be deleted.

`-url url_string` : url permitting the connection to the underlying database. Only Oracle and Sybase are supported by this command.

### **Optional arguments**

`-verbose` : turns the verbose mode on. The import tool works by default in a silent mode.

`-help` : displays the usage of the tool.

**Description** The importer deletion tool drops the importer user from the database. All databases owned by a given importer, if any, must be deleted before attempting to delete the importer user. An attempt to delete an importer owning a database will issue an error message.

**Files** None.

**See also** `o2jb_create_importer`

---

## o2jb\_delete\_schema

---

### o2jb\_delete\_schema

---

**Summary** Deletes an existing o2 schema.

**Syntax** o2jb\_delete\_schema

```
{ -url url_string |  
  -system O2 -server mysystem@myserver }  
-schema schema_name  
[-verbose]  
[-help]
```

**Mandatory arguments**

-url url\_string : url permitting the connection to the o2 database.

-schema schema\_name : name of the o2 schema to delete.

**Optional arguments**

-verbose : turns the verbose mode on. The delete schema tool works by default in a silent mode.

-help : displays the usage of the tool.

**Description** The schema deletion tool deletes an existing o2 schema. An o2 schema with associated bases can not be deleted. You must first delete the bases before calling o2jb\_delete\_schema.

**Files** None.

**See also** o2jb\_create\_base, o2jb\_delete\_base

**o2jb\_dump\_bytecode**

**Summary** Dumps the bytecode of a given class or set of classes into a `.class` file(s).

**Syntax** `o2jb_dump_bytecode`

```

    -user          user_name
    -passwd        user_passwd
    -base          base_name
    -url           url_string
    -schema        schema_name (for JB/O2 only)
    -driverclass  driver_class_name (For JB/JDBC only)
    [-package     package_name]
    [-output      output_dir]
    [-verbose]
    [-help]
    class_name*
  
```

**Mandatory arguments**

`-user user_name` : login name of an importer user created with the `create_importer` tool (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

`-passwd user_passwd` : password of the importer user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

`-base base_name` : name of the database where the bytecode is stored.

`class_name*` : list of qualified class names separated by blank. These are the classes whose bytecode is to be dumped to the corresponding `.class` output files.

`-url url_string` : url permitting the connection to the underlying database.

`-schema schema_name` : name of the o2 schema.

The following argument must be provided *for JB/JDBC only*

`-driverclass` : allows a driver Java class name to be specified.



---

## o2jb\_dump\_bytecode

---

### Optional arguments

**-output output\_dir** : name of the output directory where the `.class` files are generated. The current directory is used by default.

**-package package\_name** : dumps the `.class` files of all classes in the package `package_name` whose bytecode is stored in the database. Different packages can be passed as arguments, each with a `package` flag.

**-verbose** : turns the verbose mode on. The import tool works by default in a silent mode.

**-help** : displays the usage of the tool.

**Description** This tool dumps stored bytecode to `.class` files. Any database user is authorized to dump bytecode from the database, provided he/she has the appropriate rights to write to the output class path.

**Files** *Output files* : this tool generates `.class` files in the path given through the environment variable `$CLASSPATH` or redeclared through the optional argument `-output`.

**See also** `o2jb_store_bytecode`, `o2jb_delete_bytecode`

## **o2jb\_dump\_classes**

---

**Summary** Dumps the Java definition of all imported classes to a file.

**Syntax** `o2jb_dump_classes`

```
-user      user_name
-passwd    user_passwd
-base      base_name
-url       url_string
-schema    schema_name (for JB/O2 only)
-driverclass driver_class_name (For JB/JDBC only)
[-output  output_dir]
[-verbose]
[-help]
```

### **Mandatory arguments**

`-user user_name` : login name of a database user (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

`-passwd user_passwd` : password of the user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

`-base base_name` : name of the database whose catalog is inspected.

`-url url_string` : url permitting the connection to the underlying database.

`-schema schema_name` : name of the o2 schema.

The following argument must be provided *for JB/JDBC only*

`-driverclass` : allows a driver Java class name to be specified.

### **Optional arguments**

`-output output_dir` : name of the output directory where the output file is generated. The current directory is used by default.

`-verbose` : turns the verbose mode on. The import tool works by default in a silent mode.

`-help` : displays the usage of the tool.

---

## **o2jb\_dump\_classes**

---

**Description** This tool inspects the JB catalog and generates the definition of each imported class in the output file. This allows the user to examine which classes were imported into a given database. Renamed and hidden variables described through a configuration file are taken into account, so that the structure of each class in the output file is not necessarily the same as that defined in the corresponding `.java` file, but reflects the structure of the class instances that are stored in the database.

No update is performed to the database and any database user is authorized to run this tool.

**Files** *output file* : the output file is named after the database. The suffix `.classes` is appended to the name of the database.

**See also** `o2jb_dump_schema`

## **o2jb\_dump\_schema**

---

**Summary** Dumps the definition of all relational tables or O<sub>2</sub> classes generated for imported classes to a file.

**Syntax** `o2jb_dump_schema`

```
-user          user_name
-passwd        user_passwd
-base          base_name
-url           url_string
-schema        schema_name (for JB/O2 only)
-driverclass   driver_class_name (For JB/JDBC only)
[-output output_dir]
[-verbose]
[-help]
```

### **Mandatory arguments**

`-user user_name` : login name of a database user (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

`-passwd user_passwd` : password of the user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

`-base base_name` : name of the database whose catalog is inspected.

`-url url_string` : url permitting the connection to the underlying database.

`-schema schema_name` : name of the o2 schema.

The following argument must be provided *for JB/JDBC only*

`-driverclass` : allows a driver Java class name to be specified.

### **Optional arguments**

`-output output_dir` : name of the output directory where the output file is generated. The current directory is used by default.

`-verbose` : turns the verbose mode on. The import tool works by default in a silent mode.

`-help` : displays the usage of the tool.

---

## o2jb\_dump\_schema

---

- Description** This tool inspects the JB catalog and generates the definition of :
- all relational tables
  - or all O<sub>2</sub> classes if system\_type is O<sub>2</sub>;
- generated for the imported class in the output file.
- This allows the user to examine the database structures used to store instances of the imported classes. Knowing the database structures can be useful in two contexts:
- (1) to write complex queries through the method **where** of class **COM.ardentsoftware.jb.api.Extent**;
  - (2) to access data stored in a JB database with external applications (e.g. JDBC or SQL applications) to directly access data stored in a JB database (through a JDBC driver, for instance).
- No update is performed to the database and any database user is authorized to run this tool.
- Files** *output file* : the output file is named after the database. The suffix **.sql** is appended to the name of the database.
- See also** **o2jb\_dump\_classes**

**o2jb\_export**

---

**Summary** Takes existing O<sub>2</sub> class definitions and generates java files which are then used by java applications to manipulate O<sub>2</sub> objects.

**Syntax** `o2jb_export`

```

        -schema      schema_name
        -url         url_string
        [-d directory_name | -output directory_name ]

```

**Mandatory arguments**

**-schema** : name of the schema from which you want to export the class.

**-url** : url permitting the connection to the underlying database. **Note** : the form of this url is as follows : `o2:servername:systemname`

**Optional arguments**

**-d** or **-output** : used to specify the directory in which the java files are generated. By default the current directory is used.

Syntax : `o2jb_export -url a_Url -schema schemaName -d directoryName`

**Description** The `o2jb_export` tool takes a class from an O<sub>2</sub> schema, analyses the class structure and generates java code when possible. The O<sub>2</sub> schema and class may have been created using C++, O<sub>2</sub>C or Java. If the class to export has inheritance links, the whole hierarchy of the class is exported. You can export several classes at the same time using the following command :

```
o2jb_export -url anUrl -schema schemaName className1
className2 ...
```

To be exported, a class must fullfil certain conditions : the class to export must be empty or of type tuple. It must inherit from only one O<sub>2</sub> class which is not empty (class of type tuple) because java does not allow multiple inheritance. If it is of type tuple, it must only have attributes of types integer, real, boolean, char, string, bits.

**Files** None.

---

## o2jb\_import

---

### o2jb\_import

---

**Summary**     Import a class or a set of classes into a database.

**Syntax**     o2jb\_import

```
-user                user_name
-passwd             user_passwd
-base                base_name
-url                 url_string
-po
-schema             schema_name (for JB/O2 only)
-driverclass        driver_class_name (For JB/JDBC
only)
[-config            config_file]
[-confirm ]
[-output            output_dir]
[-classpath         class_path]
[-input             class_path]
[-nostatic]
[-import]
[-importfile]
[-array array_name]
[-nofk]             (for JB/NATIVE)
[-noextent]         (for JB/O2 only)
[-nopp]
[-verbose]
[-help]
[-package           package_name]
class_name*
```

#### Mandatory arguments

**-user user\_name** : login name of an importer user created with the `create_importer` tool (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

**-passwd user\_passwd** : password of the importer user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

**-base base\_name** : name of the database into which classes are to be imported. The database must be owned by **user\_name**. Note : The **-base** option is ignored by the JB/O2 version.

**-url url\_string** : url permitting the connection to the underlying database.

**class\_name\*** : list of qualified class names separated by a blank. These are the classes to be imported. Input classes can belong to different packages.

**-po** : option allowing the class to implement an interface `PersistentObject`

The following argument must be provided *for JB/O2 databases only*.

**-schema schema\_name** : name of the o2 schema.

The following argument must be provided *for JB/JDBC only*

**-driverclass** : allows a driver Java class name to be specified.

### Optional arguments

**-config config\_file** : name of a configuration file to be used by the import tool. An absolute or relative access path can be given together with the name, otherwise the file is searched for in the current directory.

**-output output\_dir** : name of the output directory where the patched `.class` files are generated. The current directory is used by default.

**-input class\_path** or

**-classpath class\_path** : class path searched for the `.class` files of the classes to be imported. The environment variable `$CLASSPATH` is used by default.

**-package package\_name** : a package name. All classes in the package are imported into the database. Many different packages can be passed as argument, one by **-package** flag.

**-importfile** : when set, allows the generation of import files without importing the input classes into the JB database. No connection to the database is performed and connection arguments (**-user**, **passwd** and **base**) are ignored when this option is set.

**-array array\_name** : specifies that an array must be imported. The arrays are automatically imported if they are referenced in a class. But if



---

## o2jb\_import

---

you want to assign a non-referenced array in a field of array type this option must be used.

Syntax: `-array Rectangle[] -array Rectanle[][]`

`-nofk` : specifies that the foreign keys in the relational schema must not be created. **Note:** The `-nofk` option is ignored by the JB/O2 and JB/JDBC versions.

`-noextent` : no extent will be managed for the specified imported classes or packages. If you try to import a class without extent when the superclass has an extent, a warning message is displayed.

`-verbose` : turns the verbose mode on. The import tool works by default in a silent mode.

`-help` : displays the usage of the tool.

`-nostatic` : static fields are transient.

`-import` : specifies that we want to use the non transparent version of the `o2jb_import` tool. import files are generated instead of patching the `.class` files.

`-confirm` : performs a `confirm classes` for the O<sub>2</sub> schema. **Note** : this option concerns only JB/O<sub>2</sub>.

**Description** To make a given Java object persistent, i.e. store the object in a database, its class must firstly be imported to the database.

The import tool generates and installs all database structures necessary to store/retrieve objects of a Java class into/from a database in an optimized and transparent way. In addition, the import program generates some Java source code for each imported class *C* in an *import file C.import* if the option `-import` is used. This file contains the implementation of methods declared in the interface `PersistentObject`, that every imported class must implement. Import files must be inserted into the corresponding `.java` files.

When a class being imported references another class through one of its variables, the referred class must also be imported into the same database, unless the variable is declared as transient in the configuration file. Direct and indirect superclasses are automatically imported, unless they were already imported into the same database.

[JB/O2 Description](#)

For JB/O2 version, it is not possible to import a class having an attribute with the same name as an attribute of any of its superclasses. The o2 data model considers attributes with the same name as overloaded but Java does not.

If you specify the option `-noextent`, no extent will be managed for the imported class or package. A warning message is displayed when you import a `C` class without extent, when a superclass of `C` has an extent.

It is not possible to import classes with the same name from different packages.

If you have imported a class with extent, you cannot reimport it without. You must first unimport the class.

**Files**

**config\_file** : a configuration file can be provided to customize the way classes are imported.

**input files** : the import loads the `.class` files of the input classes. These files must be found in the class path given through the environment variable `$CLASSPATH` or redeclared through the optional argument `-input`.

**import files** : an import file is generated for each imported class. The import file is named after the corresponding class. The suffix `.import` is appended to the name of the class.

**See also**

`o2jb_unimport`, `o2jb_dump_classes`

---

## o2jb\_init\_base

---

### o2jb\_init\_base

---

**Summary**      Installs the system catalog on an existing uninitialized database.

**Syntax**        o2jb\_init\_base

```
-user            user_name
-passwd         user_password
-base            base_name
-url             url_string
-driverclass driver_class_name (for JB/JDBC only)
[-verbose]
[-help]
```

#### Mandatory arguments

**-user user\_name** : login name of an importer user created with the `create_importer` tool (see note on page 5.2). **Note**: The **-user** option is ignored by the JB/O2 version. For version JB/JDBC, the values of the **-user** option must be an user with table creation privilege.

**-passwd user\_passwd** : password of the importer user. **Note**: The **-passwd** option is ignored by the JB/O2 version.

**-base base\_name** : name of the database to be initialized.  
**Note** : The **-base** option should not be used when accessing an Oracle database through a JDBC driver.

**-url url\_string** : url permitting the connection to the underlying database.

The following argument must be provided *for JB/JDBC only*

**-driverclass** : allows a driver Java class name to be specified.

#### Optional arguments

**-verbose** : turns the verbose mode on. The import tool works by default in a silent mode.

**-help** : displays the usage of the tool.

**Description**    The base initialization tool initializes an existing database in the underlying database system by installing the system catalog structure that is used by the import tool and the JB runtime system.

The user initializing an existing database must be the *owner* of the database. This is the only user authorized to import classes into the database.

The initialization tool is meant to initialize JB databases created with the `o2jb_create_base` tool. It can also be used to initialize an existing database and make it a JB database. Data already stored in the existing database will not be seen nor affected by a JB application. The data will nevertheless share the same physical volume with the JB data, i.e. Java objects stored in the same database.

**Files**           None.

**See also**       `o2jb_create_base`, `o2jb_delete_base`, `o2jb_import`

---

## o2jb\_patch\_class

---

### o2jb\_patch\_class

---

**Summary** Inserts byte code into .class files to manage persistency.

**Syntax** `o2jb_patch_class`  
`[-nopp]`  
`Class_name*`

#### Mandatory arguments

`class_name*` : list of qualified class names separated by blank. These are the classes whose bytecode is to be dumped to the corresponding .class output files.

#### Optional arguments

`-nopp` : specifies that the methods of the currently imported classes are not post processed.

**Description** The `o2jb_patch_class` tool inserts into the files resulting from java compilation (.class files) the byte code used to manage persistency and post processes the methods of the class in order to access and update persistent objects in a transparent way.

**Files** None.

**See also** None.

## o2jb\_store\_bytecode

---

**Summary** Stores the bytecode, i.e. the contents of a `.class` file, of a given class into a database.

**Syntax** `o2jb_store_bytecode`

```

    -user          user_name
    -passwd        user_passwd
    -base          base_name
    -url           url_string
    -schema        schema_name (for JB/O2 only)
    -driverclass  driver_class_name (For JB/JDBC only)
    [-classpath   class_path]
    [-package     package_name]
    [-verbose]
    [-help]
    class_name*
  
```

### Mandatory arguments

`-user user_name` : login name of an importer user created with the `create_importer` tool (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

`-passwd user_passwd` : password of the importer user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

`-base base_name` : name of the database where the bytecode is to be stored.

`-url url_string` : url permitting the connection to the underlying database.

`-schema schema_name` : name of the o2 schema.

`class_name*` : list of qualified class names separated by blank. These are the classes whose bytecode is to be stored in the database.

The following argument must be provided *for JB/JDBC only*

`-driverclass` : allows a driver Java class name to be specified.

---

## o2jb\_store\_bytecode

---

### Optional arguments

**-classpath class\_path** : class path where the `.class` files are searched for. The environment variable `$CLASSPATH` is used by default.

**-package package\_name** : stores the bytecode of all classes in the package `package_name` in the database. Different packages can be passed as arguments, each with a **-package** flag.

**-verbose** : turns the verbose mode on. The import tool works by default in a silent mode.

**-help** : displays the usage of the tool.

### Description

This tool allows the bytecode, i.e. the contents of a `.class` file, of a class or a set of classes to be stored in a database. Classes are not required to be imported into the same database. Only the database owner is authorized to store bytecode in the database.

If the bytecode of a given class is already stored, it is overwritten.

### Files

*Input files* : this tool loads `.class` files and stores their contents in the database. These files must be found in the class path given through the environment variable `$CLASSPATH` or redeclared through the optional argument **-input**.

### See also

`o2jb_dump_bytecode`, `o2jb_delete_bytecode`

## **o2jb\_test\_driver**

---

**Summary** Tests the underlying JDBC driver. **Note:** This command can be used only with the JB/JDBC version.

**Syntax** `o2jb_test_driver`

```
-url          url_name
-driverclass  driver_class_name
-user         user_name
-passwd      user_passwd
```

### **Mandatory arguments**

`-url url_name` : allows a URL to be given. Example: "jdbc:odbc:myDataSource".

`-driverclass driver_class_name` : allows a driver Java class name to be specified. Example: "sun.jdbc.odbc.JdbcOdbcDriver".

`-user user_name` : login name of an importer user created with the `create_importer` tool.

`-passwd user_passwd` : password of the importer user.

**Description** This tool tests the JDBC features according to the needs of JB and generates characteristic information of the JDBC driver on standard output.

**Files** None.

**See also** None.



---

## o2jb\_unimport

---

### o2jb\_unimport

---

**Summary**      Deletes a class or a set of classes from a database.

**Syntax**        o2jb\_unimport

```
-user            user_name
-passwd         user_passwd
-base            base_name
-url             url_string
-schema          schema_name (for JB/O2 only)
-driverclass    driver_class_name (For JB/JDBC only)
[-confirm ]
[-verbose]
[-help]
class_name*
```

#### Mandatory arguments

**-user user\_name** : login name of an importer user created with the `create_importer` tool (see note on page 5.2). **Note:** The `-user` option is ignored by the JB/O2 version.

**-passwd user\_passwd** : password of the importer user. **Note:** The `-passwd` option is ignored by the JB/O2 version.

**-base base\_name** : name of the database from which classes are to be deleted. The database must be owned by `user_name`.

**-url url\_string** : url permitting the connection to the underlying database.

**class\_name\*** : list of qualified class names separated by space(s). These are the classes to be deleted. Direct and indirect subclasses are automatically deleted.

The following argument must be provided *for JB/O2 databases only*.

**-schema schema\_name** : name of the o2 schema.

The following argument must be provided *for JB/JDBC only*

**-driverclass** : allows a driver Java class name to be specified.

**Optional arguments**

**-verbose** : turns on the verbose mode. The `unimport` tool works by default in a silent mode.

**-help** : displays the usage of the tool.

**-confirm** : performs a `confirm classes` for the O<sub>2</sub> schema. **Note** : this option concerns only JB/O<sub>2</sub>.

**Description** The `unimport` tool deletes a given class or set of classes from the database. Deletion succeeds if no other imported class (in the same database) refers to one of the classes being deleted, unless the referencing class is being itself deleted. In addition, no instance of a class being deleted or of one of its subclasses must be stored in the database, otherwise the deletion fails and an error is reported.

**JB/O<sub>2</sub> Description**

When a class with extent is unimported, the extent and all persistent instances of the class are deleted. You cannot unimport class with subclasses.

**Files** None.

**See also** `o2jb_import`, `o2jb_dump_classes`