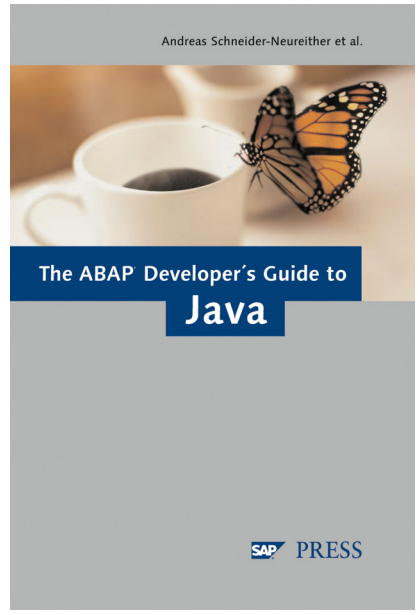
Andreas Schneider-Neureither, Bernd Noll,
Andreas Schlindwein, André Schüngel,
Dominik Wittenbeck

The ABAP® Developer's Guide to Java



SAP PRESS

Contents

Foreword

Several years ago, SAP AG announced that future application development would take place in parallel in both Java and ABAP. This paradigm shift for SAP Development was launched successfully with Version 6.20 of the SAP Web Application Server (SAP Web AS). The current release, SAP Web AS 6.40, now supports both development tracks fully and reliably, giving SAP developers a platform with an incomparable arsenal of modern and proven technologies to develop powerful, professional, and stable applications.

This twin-track development has received little attention in customer products to date, however, due in large part to a lack of information among ABAP developers, who are unfamiliar with Java and J2EE syntax and concepts, as well as their distinct implementation and integration in SAP Web AS.

This book is aimed at correcting this information deficit. Not only will it illustrate the various options that the new technologies in SAP Web AS provide—in detail and with a practical focus—but also provide experienced ABAP developers with a defined, efficient migration path to this new world of SAP development.

This book is intended primarily for accomplished ABAP developers and IT managers who want to learn how to harness the great potential of the new SAP technology and integrate it into their own IT strategies to achieve a successful outcome.

We, the authors, aren't afraid to criticize, either. As experienced developers and consultants who apply these technologies in customer systems on a daily basis, we examine the individual concepts and technologies in detail, and always with a critical eye toward their practical usability.

Writing this book—coupled with our daily consulting work—demanded a tremendous effort of the authors. This effort would not have been possible were it not for the support of both our coworkers and our families. Our special thanks go out to all of them.

We would particularly like to thank Ulrich Klingels from SAP NetWeaver Product Management, whose detailed comments enriched this book. We also want to thank Florian Zimniak at Galileo Press—who provided valuable advice, assistance, and motivation for our last book in the SAP PRESS series—for his professional support and coordination efforts and Nancy

Etscovitz at SAP PRESS for her editorial help with this translation. Our thanks also go to everyone at SNP who supported or otherwise assisted us with this book project, whether directly or indirectly. Last but not least, we thank Bernhard Hochlehnert of *SAPinfo*, who gave us the impetus for our book projects.

We trust that you will find this book to be a useful tool for familiarizing yourself with the new technologies and implementing them successfully in your projects.

We hope that this is a pleasant, stimulating read for you!

Andreas Schneider-Neureither
Heidelberg, November 2004

6 Application Layers

Conventional applications from both the ABAP and Java/J2EE worlds can be described as three-tier models. Each application contains a certain proportion of data handling, business logic, and presentation logic. In this chapter, we will examine all the layers for both sides—ABAP and Java—of the application.

6.1 Retrieval Logic and Persistence

The data retrieval logic and persistence layer is responsible for providing data to the downstream business logic, which then has to process this data. The goal of the retrieval logic for the business logic is to establish a certain amount of abstraction for the countless resources that can be accessed—including databases, files, remote calls, and services—to fully standardize access in the best case (or at least greatly simplify it) and hide the technical details that aren't relevant for the business logic.

The next sections introduce the options for dealing with the different types of resources available on both platforms.

6.1.1 ABAP

Open SQL

SQL (Structured Query Language) is used for relational databases. It is available for nearly every RDBMS (relational database management system), although in different, vendor-specific versions. The SQL standards of ANSI (American National Standards Institute) and ISO (International Standards Organization) generally serve only as guidelines, which the database vendors more or less follow.

Aside from querying data from the database, SQL also supports changes to table contents, modification of structures, configuration of user authorizations, and settings for system security. SQL is divided into DML (Data Manipulation Language) for reading and changing data, DDL (Data Definition Language) for creating and managing tables in the database, and DCL (Data Control Language) for authorization and consistency checks.

A subset of the SQL statements called *Open SQL* is implemented in all widespread database systems and is available fully in ABAP. This enables standardized access to all databases supported by SAP, making ABAP

developments nearly independent of specific database products, as long as Open SQL is used exclusively.

Native SQL

Open SQL contains only DML commands, however, which were described in detail in Section 4.1.4. In cases where these commands are not sufficient to meet a specific requirement, ABAP also permits database-specific commands. To do so, the *native SQL* statement is placed between the ABAP statements `EXEC SQL` and `ENDEXEC`:

```
EXEC SQL.
    CREATE TABLE BUILD_COMP (
             CLIENT CHAR(3) NOT NULL,
             BUILD  CHAR(9) NOT NULL,
             COMP1  CHAR(6) NOT NULL,
             COMP2  CHAR(6) NOT NULL,
             PRIMARY KEY (CLIENT, BUILD)
                         )
ENDEXEC.
```

Listing 6.1 Example of a Native SQL Statement Embedded in ABAP

The statements embedded in the ABAP coding are forwarded directly to the database system. As such, native SQL lets you use the full range of functions provided by the database-side interface.

On the ABAP side, every work process on an application server contains a database interface with a vendor-dependent layer, which hosts all communications between the ABAP side and the database.

When native SQL statements are used in ABAP programs, switching to a different database product will be costly—because database commands generally differ, you will have to find and adjust all the involved coding manually. Moreover, you should not execute any DDL operations in application programs anyway; instead, use the ABAP Dictionary to create and maintain tables. Lastly, the SAP System does not perform any additional checks of database-specific commands. For these reasons, you should avoid using native SQL in ABAP whenever possible.

Logical Databases

A *logical database* is simply an ABAP program. However, logical databases are special programs that can supply an application program with data for processing. They are most commonly used to read data from database

tables and link them with an executable program. They can also be called with function module LDB_PROCESS, which makes it possible to call several logical databases—with the correspondingly complex nesting—within an executable program.

The logical database implements database access using Open SQL access from outside of the application program. It reads the information from the database line by line and supplies it to the executable program at runtime.

Logical databases have a hierarchical organization because many tables are interrelated through their foreign keys.

Tasks

Logical databases can execute the following tasks:

- They can be used in multiple executable programs.
- They can provide a uniform selection screen for all the programs that use a logical database.
- The authorization check is saved centrally in the logical database.
- Changes aimed at boosting performance will affect all application programs that use a given logical database.

Components

The logical database is basically divided into three objects: The *structure definition* defines the data view of the logical database, the *selection* defines the user interface of the executable program, and the *database program* executes the statements for reading the data and passing it on to the calling program. The call has the following structure:

```
GET <table_header>.
...
GET <table_item>.
...
```

Listing 6.2 Theoretical Call in an Executable Program

Persistent Objects

Persistent objects belong to the *object services*, which supply applications with various central services that cannot be represented by ABAP Objects language elements directly. SAP currently provides two such object services, the *Persistence Service* and the *Transaction Service*. The Persistence Service helps ABAP developers use object-oriented data in relational databases.

Transient and Persistent Data

Data can generally be differentiated in two different categories: transient and persistent data. Put simply, *transient data* exists only during program runtime, while *persistent data* is durable, for example, in a database. Persistent data can also be found as content in the application and presentation layers. In object-oriented programs, data is usually portrayed as attributes of objects. Methods also define and use local data, of course, but we will overlook that here. In object-oriented programming, an object exists only during program runtime, between the creation and deletion of a program session. To work with persistent data in objects, accesses to the data store must be programmed within the class methods.

The logic behind using persistent objects is that the data of an object is saved in the database transparently for the developer, and is retrieved during the initialization of the object, to allow a program to continue processing the same objects that another program left behind in a certain state. Therefore, the Persistence Service is responsible for providing ways to save the attributes of an object persistently and mapping them to the correct class.

To use the Persistence Service for objects, their classes have to be defined as *persistent classes* in the Class Builder. The Persistence Service manages these objects and their states. The objects in such a class are not created using the `CREATE OBJECT` statement in an ABAP program, but instead with a method of the Persistence Service, which also ensures that the initialization is accurate. In addition to the unique ID, the persistent classes can contain key attributes to identify the object uniquely. The Persistence Service manages the persistent objects and oversees the connection between object and database.

Tutorial: Persistent Class

Objective

This tutorial shows you how to create a simple persistent class for a database table. Its objective is to show developers the details of developing persistent classes to compare them to the Java equivalent, the *Java Database Object* (JDO), later in this chapter.

Requirements

You should be familiar with the basics of the ABAP Workbench and also have had some contact with ABAP Objects.

Process

1. To create the persistent class, you use the Class Builder (Transaction SE24) or the Object Builder (Transaction SE80).
2. Enter the name for the persistent class as `ZCL_<dbtab>_PERSISTENT`.
3. In the properties for the class, be sure to select class type **Persistent Class**.

4. The created class implements the methods of interface IF_OS_STATE, which manages the object status.
5. Other classes are now generated automatically for the new persistent class: ZCB_<dbtab>_PERSISTENT and ZCA_<dbtab>_PERSISTENT.
6. In the persistence map, assign class database table <dbtab> to class ZCL_<dbtab>_PERSISTENT. To display the persistence map, choose menu path **Goto • Persistence Map**.
7. You can define the mapping for database table DBTAB here.
8. Save and activate the persistent class.
9. The following coding should give you an impression of how persistent classes are used within a context, such as a report. Reference table agent is used to assign a reference to the persistent class, ZCL_<dbtab>_PERSISTENT. Method GET_PERSISTENT is used to check whether an entry exists in the database. If no entry exists, an exception is raised. The program then attempts to create an object within this CATCH block. The entry does not exist in the database until after the COMMIT WORK. If no commit work is performed, the generated object exists only during runtime.

```
DATA: connection TYPE REF TO zcl_<dbtab>_persistent,
      agent      TYPE REF TO zca_<dbtab>_persistent.

Agent = zca_<dbtab>_persistent=>agent.
TRY.
  Connection = agent->get_persistent(
           i_key1 = wa_<dbtab>-key1
           ...
           i_keyn = wa_<dbtab>-keyn ).
CATCH cx_os_object_not_found.
  TRY.
    agent->create_persistent(
             i_key1 = wa_<dbtab>-key1
...
             i_field1 = wa_<dbtab>-field1
... ).
  CATCH cx_os_object_not_found.
    ...
  ENDTRY.
ENDTRY.
```

6.1.2 Java

To analyze the complex architectural details of the Java personality of the persistence layer on the SAP Web Application Server, we first have to examine the interdependencies of data retention within the SAP context. In the pure SAP world that you have dealt with so far, all data is saved in a centralized database. ABAP programs access the database directly, using the mechanisms described above. Why shouldn't it be as easy to implement this in Java? We would assume that new tables would be created in the ABAP Dictionary and accessed at some point in the Java coding.

While this approach may seem simple and logical at first glance, it harbors several disadvantages. The most serious of these is the fact that this approach does not comply with defined J2EE standards, as it would require the existence of an ABAP instance—which is extremely unlikely for a pure J2EE environment, which are used widely in enterprise projects outside of SAP Systems. Moreover, merging ABAP and Java tables would require Java developers to follow ABAP conventions that ensure data consistency, such as those for lock management or update requests.

Design Objectives

Nonetheless, a central database for the Java personality of the Web AS also has enormous advantages. The central instance of the Web AS builds on a central database instance that behaves similarly to the corresponding ABAP instance—Customizing and configuration data is saved along with the application tables. To enable this, SAP's design objectives pursue the following goals:

- **Strict separation of ABAP and Java persistence**
  Each personality has its own isolated database schema, characterized by two logically—or even physically—separated databases. No transaction can extend over both schemas; a Java application can access ABAP data, but not at the database level. In other words, table accesses between the ABAP and Java stacks are not possible. Instead, they are performed at the business logic level or its encapsulating middleware, for example, via Remote Function Call (RFC) by means of the Java Connector (JCo). Therefore, the collaboration has to take place at component level at this point.
- **Minimization of database administration effort**
  To keep the effort required for installation and administration to a minimum, despite the necessity of separating both database schemas, it is possible to realize both schemas within a single database. This means

an ABAP transaction accesses the ABAP schema and a Java transaction accesses the corresponding Java schema, but in the same physical database.

- **Extension of Java persistence technologies**
  Familiar features and concepts from the ABAP world, such as caching statements and support for table buffers, have been transferred to the Java world.

The object orientation of the Java language is another aspect that has had a major impact on the persistence layer architecture on the Java side. While most conventional SAP applications are still based on relational persistence and procedural code—making it relatively simple to model business data in tables—Java forces you to think in terms of objects. For this reason, SAP supports both options for accessing data, which generally must be considered separately: relational and object-based data retention. The two methods also differ in the way developers use and manipulate the data.

Open SQL for Java

Framework for a Standardized Data Access Layer

Just like Open SQL provides standardized access to databases in an ABAP environment, Open SQL for Java creates a database access layer for Java applications. This layer provides performance-boosting mechanisms, such as table buffers and statement pooling, and at the same time enables portable access to many different databases, including Oracle, IBM DB2, Microsoft SQL Server, MaxDB, and others. You don't have to modify the applications under this method, because the SQLJ subset balances out the differences between the databases, which means applications can run on different databases without requiring modification.

All the programming models that SAP covers for the supported databases are part of this *Open SQL for Java Framework*. Application developers have various options for accessing data in the persistence layer. All access options within Open SQL for Java are based on the lowest instance—the JDCB API described in Section 5.2. SAP then builds various abstraction layers on top of this programming interface—each of which exists and can be used independently of the others—and offers various functions within its own hierarchy level, each with its own advantages and disadvantages.

JDBC

As you can see in Figure 6.1, JDBC represents the lowest abstraction level. Ultimately, all the higher layers generate JDBC calls that the vendor-spe-

cific JDBC database drivers process as SQL statements—based on the Java JDBC API—and send directly to the database. JDBC's extreme popularity is due not least to the wide variety of example coding that is publicly available.

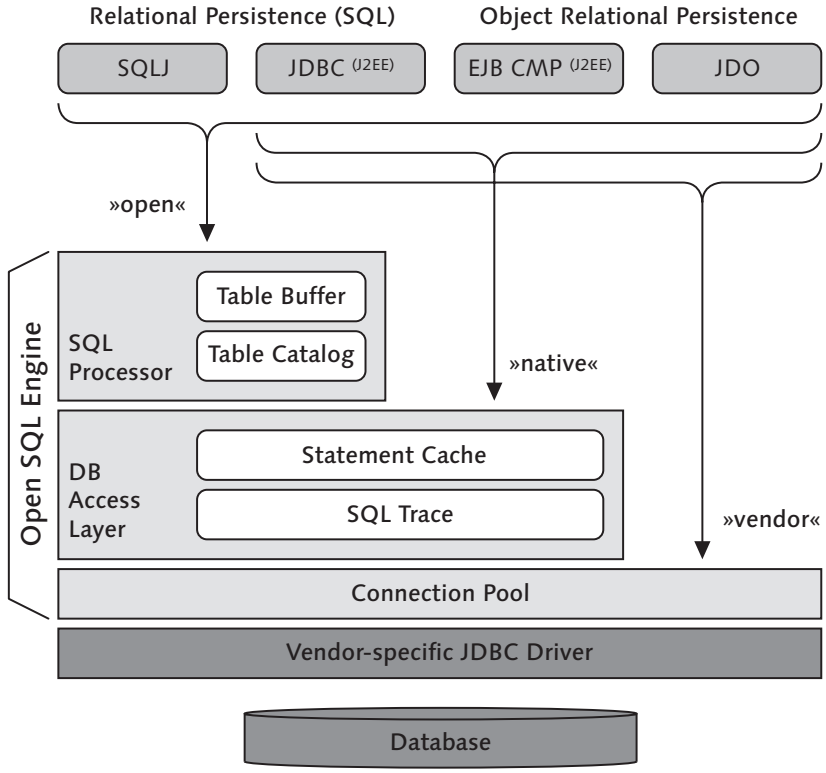


Figure 6.1 Open SQL Framework—Database Access Layers

Native JDBC

Using native JDBC alone does not guarantee database independence; however, ultimately, how the JDBC calls are executed depends on the JDBC driver implementation and the semantics of the underlying database. Note that if you use native SQL or native JDBC explicitly to implement database access, the JDBC API will not provide any framework for inspecting or validating SQL statements. Therefore, you will not be able to verify whether your application coding can be executed on other database platforms.

To avoid potential portability problems with JDBC and SQL, SAP has defined a subset of SQL statements to ensure database independence—at least for the databases that SAP supports. Here, Open SQL for Java is an

equivalent to the known Open SQL on the ABAP side, and solves the related problems associated with nearly every programming language.

The core of the Open SQL for Java framework is the Open SQL Engine, which consists of three layers that build on one another, with each higher layer providing more functions. The lowest layer is the connection pool, which is the foundation for the database access layer, which, in turn, is the foundation for the top layer—the SQL processor layer (see Figure 6.2).

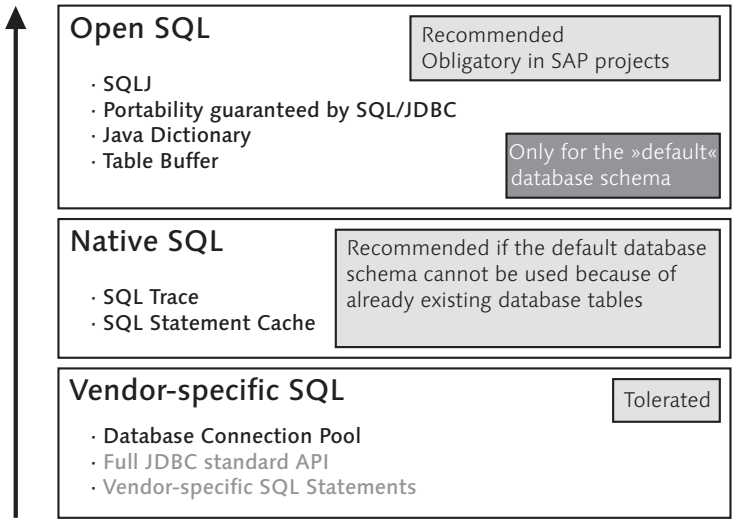


Figure 6.2 Open SQL Engine—Layer Model

Relational and Object/Relational Persistence

As mentioned above, SAP supports various programming models for accessing data. You can differentiate between relational and object/relational persistence and choose between different approaches for implementing each model. Within the relational model, we differentiate further between the SQLJ and JDBC persistence scenarios. For object-oriented persistence, Java provides two possible implementations, using either enterprise entity beans or Java Data Objects. The individual programming models are described in detail later in this chapter. For now, all you need to know is that every model can be built on any of the three layers within the Open SQL layer model (aside from SQLJ, which is based on the highest layer of the Open SQL Engine), which means a connection to the lowest layer is an essential prerequisite. Accordingly, SAP separates these groups into "open," "native," and "vendor-based" connection models. All three layers of the Open SQL Engine are described below.

Connection Pool – Vendor SQL

The lowest layer is the connection pool, which builds directly on the vendor-specific JDBC driver. As you are already aware, creating and providing database connections is a complex process, and an expensive undertaking—from a system resource perspective. The connection pool saves connections to the same data sources in a *pool,* allowing you to create connections without any time delay. The pool also enables access to the default database schema, which is already predefined in the connection pool and does require adjustment.

Once administrators create the connection pools centrally on the J2EE server, the pools are referenced through only logical, unique names from the Java Naming and Directory Interface (JNDI) context. This keeps sensitive data, such as authentication data or maximum loads, away from developers, and prevents these typically architecture-based parameters from being shifted to the application logic.

Applications declare resource references to the pool as a data source. They receive and return their connections through the pool. Thus, the connection pool is shared by both different requests and different applications. Aside from the performance aspects mentioned above, this approach also provides you with a central repository where you can both configure and monitor database connections and accesses.

As you can see in Figure 6.3, each connection approach at least builds on this layer, which means all the functions in the layer are always available.

Vendor SQL or JDBC

When applications access data in a relational database directly, JDBC is used. Although using abstraction models such as enterprise entity beans or JDOs hides this fact from the developer to a certain extent, these object-oriented models also rely on JDBC. Therefore, if you build on this lowest level, you can use the proprietary capabilities of the individual databases, but will lose all the benefits afforded by Open SQL—portability, table buffering, and SQL statement cache, to name just a few. Because this approach involves working directly at the vendor-specific layer of the database, SAP calls this approach "vendor-specific" or "vendor SQL/JDBC."

SAP specifies persistence based on this third layer as "tolerated." Therefore, you should develop at this level only if your applications absolutely require the database's proprietary capabilities.

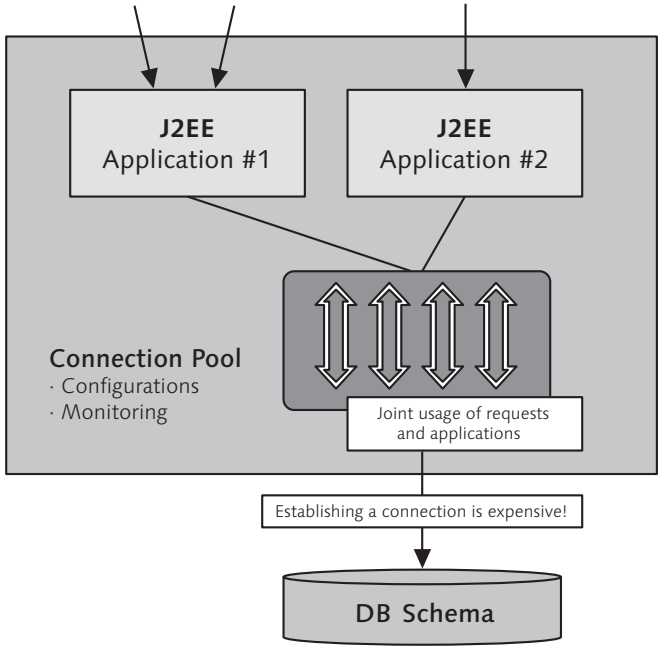


Figure 6.3 Connection Pool

DB Access Layer—Native SQL/JDBC

The second layer for database access builds on the connection pool layer—in accordance with the layer model—to enhance certain functions. Like the vendor-specific layer, this layer gives developers all the functions of the underlying proprietary database system, but again at the cost of portability and table buffering.

Native SQL or JDBC

All method calls are sent to the underlying JDBC driver directly and unchanged. Basically, the implementation of the Native JDBC API is a simple wrapper around the vendor-specific JDBC driver, but with two decisive enhancements regarding the speed and ease of maintenance of the J2EE engine—SQL trace und statement pooling.

SQL Trace

The SQL trace is available on demand to trace all SQL statements submitted to the database and executed using methods of this layer or the higher layer of the Open SQL Engine—the SQL processor layer. You can activate and deactivate the SQL trace dynamically in the Visual Administrator. The trace format is database-independent. Aside from the actual SQL statements, the log entries contain information about the time of a statement, its duration, its input parameters, and its results (where relevant), along with context information.

The SQL trace is available through a browser interface within the SAP Web AS and is particularly helpful for performance analyses. It reveals the causes of errors and poor persistence designs quickly and easily, especially the higher-level APIs that are used create an unreasonably large set of SQL statements. The SQL trace can also be a big help at development time, as it shows developers which SQL coding is generated from their JDOs, JSPs, servlets, and EJBs.

Statement Pooling

Statement pooling improves runtime performance by caching frequently used SQL statements. A buffer helps the engine detect whether an identical request has been recently placed. You can save a significant amount of CPU time because frequently used requests have to be prepared only once (in the prepare phase) and can then be executed repeatedly, which reduces the total number of parse routines sent to the database.

```
PreparedStatement ps = con.prepareStatement("SELECT *
FROM ZRM_RES_MASTER WHERE RESID = ?");
ps.setInt(1, 256);
[...]
ResultSet rs = ps.executeQuery();
[...]
ps.close();
```

Listing 6.3 Life Cycle of a SQL Statement

You should note that this source code fragment can be part of a servlet that is executed several times within the J2EE application, but with different `empl_id` values. The resulting SQL statement would have to be prepared and sent to the database each time that the servlet is executed. Preparing the SQL statement—which means the database has to parse it and create an optimized execution plan—is an extremely cost-intensive process on most systems, and represents a significant performance overhead in the long term.

Prepared-Statement Object

Statement pooling lets the application reuse a statement object that is already prepared (`PreparedStatement` object), similar to the approach of reusing database connections under connection pooling. This reuse is completely transparent for the application. When an application uses a `PreparedStatement` object, whether or not that object participates in statement pooling is immaterial. No coding changes are required. When an application closes a `PreparedStatement`, it can reuse it again with the `Connection.prepareStatement()` method.

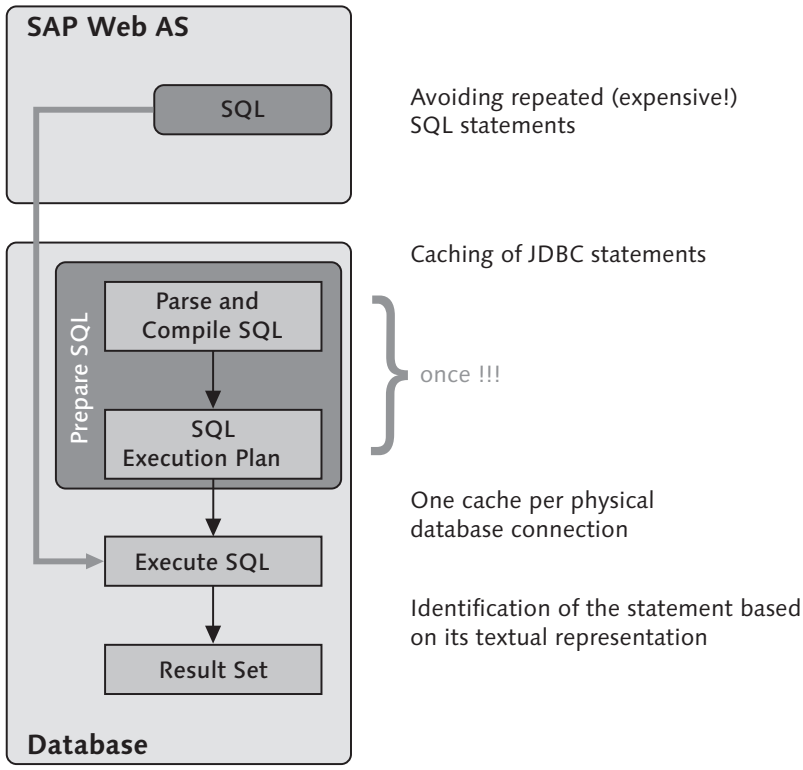


Figure 6.4 Statement Cache

A statement pool instance is associated with a physical database connection and caches `PreparedStatement` and `CallableStatement` objects that are created for this connection. Each time a `prepareStatement()` or `prepareCall()` method is called for a specific connection, the native JDBC driver automatically searches the associated statement pool for a suitable statement. The following criteria are relevant for the developer:

- The statement text must be identical to the statement in the cache (case-sensitive).
- The call type must be the same (`prepared` or `callable`).
- The scrollable type of the result set of the call must be the same (`forward-only` or `scrollable`).

If a suitable statement is found in the pool, a new `PreparedStatement` object is created and passed on to the calling program. Otherwise, the prepare call is parsed initially to create a new object. Each of these new objects is pooled when its `close()` method is called.

SQL Processor—Open SQL/JDBC

The third and highest layer of the Open SQL Engine is the SQL processor layer. It manages the table buffer, which is another building block for boosting efficiency. The objective of the table buffer is to hold parts of the database tables on the application server after they are first accessed in order to avoid multiple accesses to the same datasets within the database. This reduces the load on both the database and the network. A buffer exists for each database schema and Web AS instance, although one buffer can work for several connections simultaneously.

This buffering can be configured for each table individually, and you can also configure the buffering granularity—to buffer only some of the table contents or the entire table. Buffering is transparent for the application: The first buffer access loads the data into the buffer implicitly so that subsequent accesses are served directly from the buffer and don't have to access the database.

Visual Administrator

You can display statistics regarding table buffer usage in the *Visual Administrator,* in the **Monitoring Services** tab.

While the native SQL/JDBC approach, which builds on the second layer of the Open SQL Engine, should be selected only if no standard database schema can be used (because data tables already exist, for example), the Open SQL approach is generally the first choice. The roles at development time are clearly structured in this model, and the Java Dictionary is included in the process explicitly.

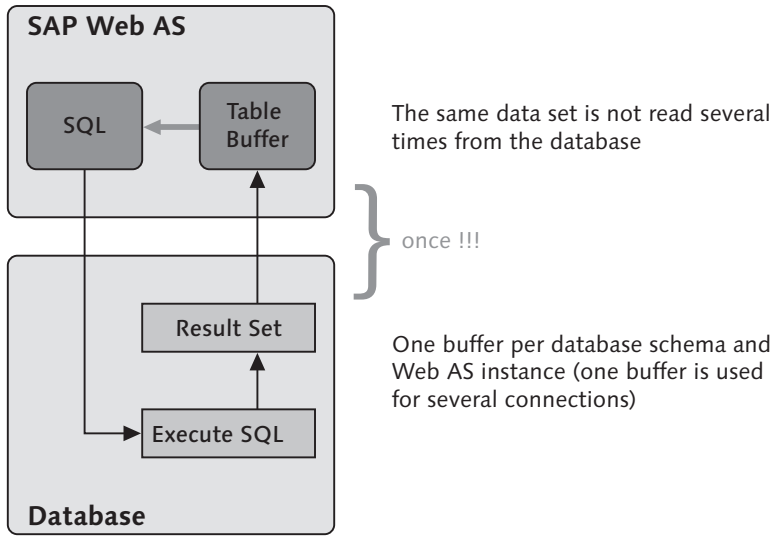


Figure 6.5 Table Buffer

Fully integrated in SAP NetWeaver Developer Studio, the Java Dictionary is used to manage the life cycle of the database object, that is, definition, creation, and modification. It is available only to those developments passed on the top layer of the SQL engine. As is true for ABAP, DDL operations should be executed only within the Dictionary.

Requests and DML expressions are handled by the Open SQL Engine, which performs a dual role in this model. It also handles all database connections, SQL statement processing, table buffering, statement pooling, and SQL trace.

In addition to these performance optimization features, the SQL processor level rounds out the concept of the Open SQL Engine by offering functions that make a new programming model possible within the Java persistence. Ultimately, all of the programming models introduced so far (JDBC, EJB, JDO)—regardless of which layer they are built on within the Open SQL Engine—use the JDBC driver to generate the finished SQL statements.

It's also evident that the Open SQL Framework for Java is generally quite similar to its ABAP counterpart, but with one major difference: The syntax of pure JDBC requests (including derived requests) cannot be analyzed until runtime; therefore, errors cannot be detected at design time, making the overall development process more complex. To avoid these difficulties, SAP has developed another abstraction level for persistence—SQLJ.

SQLJ

SQLJ defines a syntax for embedding static SQL expressions in Java source code—in contrast to JDBC, where SQL statements are passed on as string arguments of a JDBC method.

Because the Java compiler cannot handle these expressions, source code files with SQLJ elements are saved with file extension *.sqlj*. An SQLJ translator in the Open SQL processor replaces these elements with calls to the SQLJ runtime environment in a preprocessor step. It is then possible to compile the resulting Java source text.

SQLJ was initiated by Oracle, which founded an SQLJ consortium that grew to include Oracle, IBM, Microsoft, Sun, Sybase, Tandem, and Informix. Oracle then developed the reference implementation and standardized it as an ISO/IEC specification (number 9075–10).

The API used for SAP's Open SQL for Java Framework was derived from this specification. This SAP implementation ensures that the syntax is always compliant with the Open SQL grammar, which results in maximum database portability—since the Open SQL syntax is a subset of the SQL syntax that all leading database vendors support. Consequently, you cannot use SQLJ to process any database-specific SQL calls.

Open SQL Grammar

The Open SQL grammar is based on Entry Level SQL and is specified by ISO/IEC 9075 (Third Edition, November 1, 1992). It also supports the following SQL constructs:

- Joined tables
- Dynamic parameter specification

Open SQL supports the following subset of SQL statement sets:

- Queries
- Data Manipulation Language (DML)

Syntax

The SQLJ syntax is extremely simple to read: In SQLJ, SQL expressions start with the directive `#sql`. The precompiler skips the Java coding and processes only the SQL coding directly, with an initial syntax check. If no errors are found, the translator generates the Java source code, converting the SQLJ expressions into the necessary JDBC calls.

The SQLJ translator is seamlessly integrated in SAP NWDS. When the SQLJ source files are saved, the corresponding Java classes are generated automatically. The original SQLJ source files are displayed and edited to debug the application.

The objective of this API, which is defined at a higher level than JDBC, is to create simpler, more compact, more robust programs. Although the programs are certainly more robust—because syntax, semantics, type validity, and portability are checked at development time, and not after deployment at runtime—we cannot say that the SQL commands and source coding have become much less complex.

Nor is the test phase necessarily shortened, because the additional precompiler cycle shifts this phase forwards to before the deployment operation. SAP is working to mitigate these effects, however. Because the development environment supports the resolution of SQLJ statements, you can already test at design time.

The code is not any less complex; in fact, it can now be even more confusing due to the resulting mixture of languages and syntaxes. For example, variables are addressed in SQLJ with `:varName`.

While the SQLJ model represents a simplification for practiced ABAP developers, it is similar to the embedding of Open SQL in ABAP coding. But, because the SQL statements are hard-coded in the Java source text, and the syntax check doesn't support dynamically generated statements, SQLJ lets you use only static SQL functions contrary to Open SQL in ABAP, which, to a certain extent, supports the dynamic generation of SQL statements.

SQLJ Development

SQLJ statements start with directive `#sql` and finish with a semicolon. In addition to the reserved Java keywords, the `iterator`, `context`, and `with` keywords are also reserved within SQLJ expressions. Like Java coding, SQLJ statements are case-sensitive.

The actual SQL statement is contained within curly brackets, `{` and `}`, and is case-insensitive. Host variables have a colon (`:`) as a prefix.

```
#sql context Ctx with (dataSource = "jdbc/SYS");
String var;
#sql [ctx] { Select col into :var FROM tab };
```

You can use the following comments within an SQLJ source file:

- Java-like comments (`/* ... */` or `//`)
- SQL-like comments (`/* ... */` or `--`)

The SQL comments can be used for only the SQL parts of the source coding; you have to use Java comments outside of the SQL fragment:

```
/* #sql context Ctx with (dataSource = "jdbc/SYS");*/
// String var;
#sql [ctx] {
       --Select col into :var FROM tab
};
```

Host Variables

Java host variables are used to exchange data between Java (the host language) and SQL (the embedded language). They have the following syntax:

```
<host expression> ::= (IN | OUT | INOUT)?
  ':'( <java variable> |
       '(' <java expression> ')' ).
```

Host variables and expressions can be used anywhere in an embedded SQL statement where the Open SQL grammar permits the use of dynamic parameters. To use a Java variable as a host variable, you must preface it with a colon (:). Moreover, the variable names within the Java part must be identical to the names in the SQLJ part of a source file, including case sensitivity:

```
String res_id = "1";
       #sql [ctx] { DELETE FROM ZRM_RES_MASTER
                    WHERE RESID = :res_id };
```

Host Expressions

Like variables, complex Java expressions can also be embedded in SQL statements as host expressions. Host expressions must be enclosed with `:(` and `)`. Host expressions are evaluated from left to right as they appear within the statement.

In the following coding example, two Java expressions are embedded in a SQL statement: `ref.getKey()` is an IN parameter, while `values[++i]` is an OUT parameter. Both expressions are evaluated after the statement is executed.

```
String[] values = new String[5];
MyClass ref = new MyClass();
int i = 3;
#sql [ctx] { SELECT col
                        INTO :(values[++i])
                        FROM dbtab
                        WHERE key = :(ref.getKey()) };
```

Parameter Mode

To determine the parameter mode of a host variable or expression, you can use one of the optional parameter mode indicators "IN," "OUT," or "INOUT" (all directions are from the database perspective). While this aids comprehension of the source coding, the system actually recognizes and executes the data flow automatically. The IN parameter indicates that data is passed from the Java variable to the SQL statement, while the OUT parameter shows that the result of the SQL statement is passed back to the Java application. INOUT defines a data flow in both directions, as illustrated in the following source text:

```
#sql [ctx] { SELECT col
                        INTO :OUT var
                        FROM dbtab
                        WHERE key = :IN (ref.getKey()) };
```

You should exercise caution when using host expressions, as they are evaluated at specific times: OUT expressions are evaluated after the SQL statement is executed, while IN expressions are evaluated beforehand.

Database Connection Context

Database connections are identified in SQLJ through a defined *connection context*, which specifies the target database, the session, and the transaction. All SQLJ expressions or DML statements have to use an explicit connection context. This means such expressions have to contain a label to determine the connection context object where the expression is executed. Simply put, the connection context object represents a database connection.

The SQLJ translator substitutes this connection context declaration with the declaration of a specific Java connection context class, which implements the `sqlj.runtime.ConnectionContext` interface. Because the generated class contains static variables, a connection context can only be declared as a global or static inner class.

The connection context class, in contrast to the object, does not represent a database connection, but instead a data source and a logical catalog (at design time); the latter is discussed later in this section.

We differentiate between two types of data source connection contexts: The URL connection context has constructors that make it possible to instantiate a new connection context based on a URL. In contrast, the data source connection context makes it possible to create an object based on a data source.

Connection Context with Data Source

The declaration of a connection context can contain a `with` clause that specifies the value for the data source. This case involves a connection context with data source: The data source is linked and can be found under the specified name in the JNDI directory. The default constructor generates an instance of this context class, which contains a JDBC link to the associated data source. If a `with` clause is not specified, a URL connection context is involved. The following coding illustrates a connection context with data source:

```
#sql context SysCtx
with (dataSource = "java:comp/env/jdbc/MyDB");
 [...]
       SysCtx sysCtx = new SysCtx();
       #sql [sysCtx] { DELETE FROM dbtab WHERE key = 17 }
;
```

```
[...]
      sysCtx.close();
```

SQLJ in the IDE The SAP NetWeaver Developer Studio models the entire development process of the persistence of a Web AS J2EE application.

Java Dictionary Before you create a new project, all the tables you need must be defined in the Java Dictionary. The Java Dictionary is integrated completely in the NetWeaver Developer Studio. When you create new tables, metadata for the tables is initially created only on the client side (the developer's workstation) and then generated in the respective database during the deployment process. Because the procedure for developing with the Java Dictionary was described in detail in the corresponding tutorial in Chapter 5, we will only address the major SQLJ issues here.

Tutorial: SQLJ Development

Objective You have to create the SQLJ source files. You can either create completely new files or convert existing pure Java source code into SQLJ source code and them embed SQL statements within them. As described above, the SQLJ translator generates Java classes from the SQLJ files automatically as soon as you save your work. For this reason, you should always use the SQLJ layer, and never the Java files directly, as you will otherwise create inconsistencies because the Java classes will be overwritten the next time the corresponding SQLJ files are saved.

Requirements

- You are working in the SAP NetWeaver Developer Studio.
- A project already exists.

Process Use the wizard to create new files:

1. Choose **File · New · Other** ...
2. Select **Persistence** on the left side of the screen and then **SQLJ Source** on the right side.
3. Choose **Next**.
4. Enter the required information, as you would do so for a Java file.

To convert an existing Java source file, proceed as follows:

1. Create a Java source file—if you have not done so already or do not have a usable file.
2. Click on the Java source file with the right mouse button and choose **Convert to SQLJ** from the context menu.

Result

You now have a SQLJ file and a Java file with the same name. Never edit the Java file, as it contains only the generated code.

Validation—SQLJ Checker

The SQLJ Checker is integrated seamlessly in SAP NetWeaver Developer Studio. When the SQLJ files are converted—which is performed automatically as soon as the data is saved—the checker runs through the embedded SQL statements. The checks validate compliance with the Open SQL grammar and also test the schema against an offline catalog provided by *.gdbtable* files. These files were created when you created the Java Dictionary project and contain all the metadata, which was also used in the deployment process.

To perform this schema check, the SQLJ converter has to know the path of the *.gdtable* file that contains the offline catalog. Therefore, you must associate this file with your project:

Procedure

1. Select the project.
2. Select **Properties** in the context menu.
3. Select **SQLJ Translator**.
4. Select **XML Source** and enter the path for the *.gdtable* file.
5. Click on **OK**.

Once you associate the offline catalog description, the SAP NWDS allows you to localize SQL errors at design time. Conversion errors and Java compiler errors are displayed directly. Options are also available to navigate directly to the relevant part of the SQLJ source file, as well as set breakpoints in it.

Debugging

Although you should not use debugging on the pure Java code that is generated, you can activate it for the SQLJ source files at any time. Otherwise, the debugging is exactly like debugging for Java source files: Breakpoints are set within the Java files, the source text is analyzed step by step, and the values are checked. You cannot set breakpoints for SQLJ statements.

Procedure

To activate SQJL debugging, proceed as follows:

1. Choose the menu path **Window · Customize Perspective …**
2. Select **Other**.
3. Select **SQLJ Debugging**.
4. Click on **OK**.
5. SQLJ Debugging is added to the **Run** menu.
6. Choose **Run · SQLJ Debugging**.

Result SQLJ debugging is now active for the current session.

The procedure for writing the actual SQLJ source texts almost always involves the following steps:

1. Declare a database connection context object, for example:

   ```
   #sql context SysCtx with (dataSource = "jdbc/myDB");
   ```

   This object is based on the connection context class.
2. Create a connection to the database by instantiating the object (connection context).
3. You can now use this connection to send SQL statements and process the results.
4. Close the connection.

Combination of SQLJ and JDBC

To develop dynamically generated statements, you can use JDBC on the relational persistence side to implement dynamic SQL requests and statements. Because SAP recommends using Open SQL (SQLJ) to implement the Java persistence—and even requires it explicitly for internal developments—you may ask how you can combine these two models.

You can combine SQJL and JDBC to use dynamic and static expressions together in the same application. JDBC connections and SQLJ connection contexts are mutually convertible, as are SQLJ iterators and JDBC result sets.

Exchange Connections Because Open SQL and SQLJ are converted to JDBC requests at runtime, by using the Open SQL Engine, both of them can employ the same database connection and transaction. Conversely, SQLJ cannot use the same connection or transaction as native SQL/JDBC or vendor SQL/JDBC, because the latter don't run the full stack of the Open SQL Engine.

Using the JDBC Connection with SQLJ All connection context classes have a constructor that contains an existing JDBC connection as an argument. The SQLJ connection created using this constructor shares the underlying database connection with the JDBC connection from which it was created. When the SQLJ connection context is closed with the `close(boolean closeConnection)` method, the underlying JDBC connection is also closed. If the Boolean value `true` is passed on an argument, however (or, for better readability, as constant `ConnectionContext.KEEP_CONNECTION`), the `close()` method call

merely disassociates the SQL connection context object from the underlying JDBC connection, which means the former is not closed.

In the following coding example, an SQLJ connection context, `ctx`, is created by the JDBC connection `conn`. `ctx` and `conn` now share the same database connection. The `INSERT` and `DELETE` statements are both performed for this connection and share the same transaction.

```
#sql context MyCtx;

//...

Connection conn = ... ;
Statement stmt = conn.createStatemnt();
stmt.executeUpdate( "INSERT
    INTO ZRM_RES_MASTER
      (MANDT, RESID, RESTYPE, DESCRIPTION,
      INV_NUMBER, LOC_ADDRESS)
  VALUES (100, 1, 'R', 'Conference room 1st floor',
                  null, '0000100100')");

MyCtx ctx = new MyCtx(conn);

#sql [ctx] { DELETE FROM ZRM_RES_
MASTER WHERE RESID = 1 };
```

Listing 6.4 Connection Sharing from JDBC to SQLJ

JDBC Connection from SQLJ Context

The `getConnection()` method of the `ConnectionContext` interface makes it possible to receive a JDBC connection from an underlying SQLJ connection context. In the following example, the JDBC connection that underlies the `ctx` SQLJ connection context is made available to the pure Java coding, outside the SQLJ coding. `ctx` and `conn` then share the same database connection. The `INSERT` and `DELETE` statements are both performed for this connection and share the same transaction.

```
#sql context DemoCtx with (dataSource = "jdbc/DEMO");

// ...

DemoCtx ctx = new DemoCtx();
#sql [ctx] { INSERT
  INTO ZRM_RES_MASTER
```

```
    (MANDT, RESID, RESTYPE, DESCRIPTION,
     INV_NUMBER, LOC_ADDRESS)
  VALUES (100, 1, 'R', 'Conference room first floor',
                  null, '0000100100') };

Connection conn = ctx.getConnection();
Statement stmt = conn.createStatemnt();
stmt.executeUpdate(
       " DELETE FROM ZRM_RES_MASTER WHERE RESID = 1");
```

Listing 6.5 Connection sharing from SQLJ to JDBC

Exchanging Result Sets/Iterators

Result sets and iterators can be shared—like database connections—and are mutually interchangeable.

JDBC Result Set to SQLJ

It's easy to convert a JDBC result set to an SQLJ iterator, using an SQLJ `CAST` statement. You can apply the `CAST` statement to any result set iterator in the current viewing area in Open SQL/SQLJ. To ensure compatibility with SQLJ translators from other vendors, you should apply only the `CAST` statement to public result set iterators. Once the SQLJ `ResultSetIterator` object has been created, you should use the results of this method to handle all data retrieval operations.

In the following example, JDBC result set `rs` is converted to an SQLJ result set iterator with the `CAST` statement.

```
#sql iterator NamedIterator (String name);

//...

NamedIterator namIter;
Connection conn = ...

Statement stmt = conn.createStatement();
ResultSet rs =
       stmt.executeQuery("SELECT RESID FROM
                          ZRM_RES_MASTER"
);

#sql namIter = { CAST :rs };
```

```
while (namIter.next()) {
       System.out.println(namIter.name());
```

Listing 6.6 Converting a JDBC Result Set to a SQLJ Result Set Iterator

Iterators from SQLJ to JDBC

You can use SQLJ results records within JDBC in a similar manner. Every `ResultSetIterator` object has the `getResultSet()` method, which is used to retrieve the underlying JDBC `ResultSet` object. As this example once again clearly demonstrates, SQLJ merely hides the underlying JDBC layer from the user.

Conversely, once the JDBC `ResultSet` object is created, you should use this specific object instance to transfer the data to the surrounding Java program—instead of going to the additional (redundant) effort of instantiating a SQLJ `ResultSetIterator`.

In the example below, the `getResultSet()` described above is called for the SQLJ `ResultSetIterator` object, `namIter`, and returns it as the JDBC `ResultSet`.

```
#sql iterator NamedIterator (String name);

//...

NamedIterator namIter = null;
#sql [ctx] namIter = { SELECT RESID FROM
                       ZRM_RES_MASTER };

ResultSet rs = namIter.getResultSet();
while (rs.next()) {
       System.out.println(rs.getString(1));
}
```

Listing 6.7 JDBC Taking Over a SQLJ Iterator

Object/Relational Persistence

In the previous chapters, we mentioned the object orientation of the Java language several times, but didn't elaborate on any details of Enterprise Java Beans. Now, we'll examine the entity beans more closely.

Subjects of Business Processes

Entity beans model business concepts that can be expressed as subjects. This general model helps developers decide whether a business concept is suitable for implementation as an entity bean. Unlike session beans, entity beans aren't business processes; rather, they are business objects or

actual entities. They describe the state and the behavior of objects in the real world, and enable developers to encapsulate the data and business rules that belong to a specific concept. Therefore, these beans represent data in the database, which is why changes to the beans automatically result in changes to the database.

There are many advantages to using entity beans instead of accessing the database directly. The data is molded into object form, providing a simple mechanism for accessing and changing it, which a method employed by the beans themselves. In a sense, the developer doesn't communicate with the database, but instead, he or she communicates with objects by using the method, `PersonObject.tellMeYourName()`. If used properly, this method simplifies the implementation and makes the coding easier to understand (think of the countless SQL statements, many of them nested, that you would otherwise have to deal with). It also increases your chances of writing reusable software. However, you must ensure that an entity bean holds all the functions to ensure data consistency and simplicity for the developer.

When a new bean is generated, a new data record has to be added to the database, and a bean instance linked with this data. If the bean is used and its state changes, these changes have to be synchronized with the data in the database: adding, changing, or deleting entries. Therefore, the communication between the application and the database still takes place, but is transparent to the developer. This communication process of coordinating the database with the data represented by a bean instance is called *persistence*.

We differentiate between two types of entity beans, which implement this persistence using two different concepts: container-managed persistence and bean-managed persistence.

Container-Managed Persistence (CMP)

Under container-managed persistence, as the name implies, the persistence is managed automatically by EJB containers. These containers know how the bean's instance attributes are mapped to the database (or the table fields within it) and take care of inserting, changing, and deleting the data for the entities in the database.

Developer Perspective

From the developer's perspective, CMP entity beans are easier to program, because they enable you to focus on implementing the business logic by delegating responsibility for persistence to the container. When

you implement a bean of this type, you define a mapping to specify which fields the container will manage and how they are mapped to the database. Once defined, the container generates the necessary logic to save the state of the bean instance automatically.

Fields that are mapped to the database are called *container-managed fields,* and can contain any primitive Java types or serialized objects. The advantage of CMP is that the bean can be developed independently of the underlying database that saves its state later. Container-managed beans can be used in both relational and object-based databases. The bean's state is defined independently, increasing flexibility and therefore, possibilities for reuse.

A general disadvantage of CMP is that it requires complex mapping tools to define how the fields are mapped to the database. In some cases, however, it will suffice to map each field in the bean to a column in the database or serialize it in a file. But, there are also much more complicated cases; for example, a bean's state could be defined based on a complex relational database join. Even so, the SAP NetWeaver Developer Studio features many different functions for defining the mapping.

O/R Mapping

SAP calls this type of mapping *O/R mapping* (object/relational mapping). This concept involves certain O/R mapping rules that determine which Java data types can be mapped to which JDBC types. If you create the O/R mapping in the SAP NWDS, these requirements are fulfilled automatically; if you deviate from this schema, the Developer Studio also features O/R mapping verification.

O/R Mapping Rules

Enterprise Bean Requirements

Each entity bean class corresponds to a separate table in the database. To ensure data integrity, you cannot map different bean classes in the same table.

Rules for the CMP Fields

A CMP field, which represents a basic attribute, is mapped to a single column. The following JDBC types are accepted for the corresponding CMP fields:

Java Data Type	Possible JDBC Data Types	Default JDBC Data Type
`java.lang.String`	VARCHAR, CHAR, LONG-VARCHAR, CLOB	VARCHAR
`byte[]`	VARBINARY, BINARY, LONG-VARBINARY, BLOB	VARBINARY
`java.lang.Byte[]`	VARBINARY, BINARY, LONG-VARBINARY, BLOB	VARBINARY
`Short`	SMALLINT	SMALLINT
`java.lang.Short`	SMALLINT	SMALLINT
`Int`	INTEGER	INTEGER
`java.lang.Integer`	INTEGER	INTEGER
`Long`	BIGINT	BIGINT
`java.lang.Long`	BIGINT	BIGINT
`Float`	REAL	REAL
`java.lang.Float`	REAL	REAL
`Double`	DOUBLE, FLOAT	DOUBLE
`java.lang.Double`	DOUBLE, FLOAT	DOUBLE
`java.math.BigDecimal`	DECIMAL, NUMERIC	DECIMAL
`java.util.Date`	TIMESTAMP	TIMESTAMP
`java.sql.Date`	DATE	DATE
`java.sql.Time`	TIME	TIME
`java.sql.Timestamp`	TIMESTAMP	TIMESTAMP
`java.sql.Clob`	CLOB	CLOB
`java.sql.Blob`	BLOB	BLOB
`Boolean`	SMALLINT	SMALLINT
`java.lang.Boolean`	SMALLINT	SMALLINT
`Byte`	SMALLINT	SMALLINT
`java.lang.Byte`	SMALLINT	SMALLINT
`java.io.Reader`	VARCHAR	VARCHAR
`java.io.InputStream`	VARBINARY	VARBINARY

Table 6.1 Rules for Mapping CMP to JDBC

Relationships are implemented as references between primary key columns and foreign key columns.

Rules for Reference Fields

You define one or more different foreign key column(s) for each relationship. If `n` connections exist between two beans, then `n` mappings have to exist between the primary and foreign key columns as well. The foreign key and primary key both must have the same JDBC data type.

In addition, a column with type "unique key" cannot be part of a foreign key. For that reason, you cannot define a foreign key column as "unique," because the corresponding primary key would be "unique" automatically.

When you implement a 1:1 relationship, the foreign keys are contained in one of the two tables involved in the relationship.

1:1 Relationship

In a 1:n relationship, the foreign keys are located in the table that belongs to the bean, which represents the `n` side of the relationship.

1:n Relationship

To define an n:m relationship, you have to implement an intermediate table that contains the foreign keys for both primary keys of the objects involved in the relationship. The columns must have the same JDBC type as the primary key columns.

n:m Relationship

The validation functions in O/R mapping cannot detect and handle the following errors:

Restrictions

- A column is defined as a logical foreign key, but is not a true foreign key.
- A column is a primary key, but is defined as a foreign key.

CMP is often referred to as *declarative persistence*. It is very easy to use, even if the object model of the persistent data is complex. You don't have to program any SQL statements—you can generate the O/R mapping, the corresponding tables, and the SQL statements automatically within the development environment.

Tutorial: Creating a Container-Managed Entity Bean

This tutorial describes the procedure for using the wizard in the SAP NetWeaver Developer Studio to create an entity bean. You can also create enterprise beans, using the context menus of the relevant project.

An EJB module project already exists.

Requirements

1. Choose the menu path **File · New · Other**.
2. Choose **J2EE · EJB** on the left side of the first wizard page and then **Enterprise Bean** on the right side.

Process

3. Click on **Next**.
4. Enter a name for the new entity bean in the **EJB Name** field.
5. Select the name of the project where you want to create the bean in the **EJB Project** field.
6. Choose **Entity Bean** in the **Bean Type** field.
7. Specify a package in the **Default Package** field or, if none exists yet, create a new one.
8. Choose **Generate default interfaces** or specify which interfaces will be generated and used, as shown in Figure 6.6.

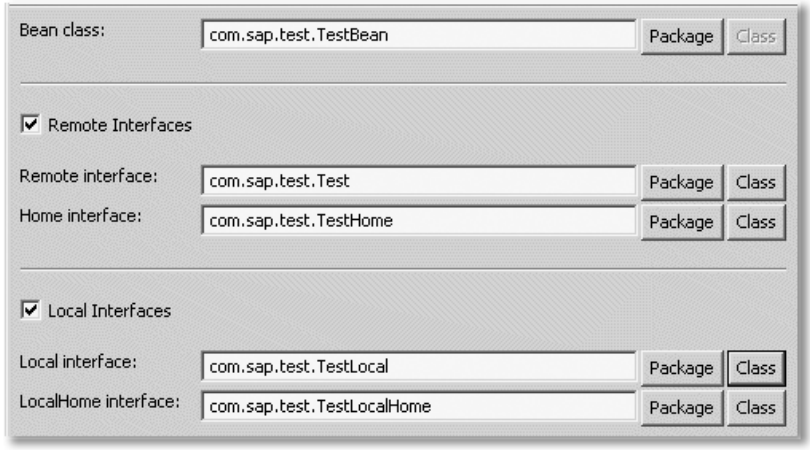


Figure 6.6 Selecting Bean Interfaces

9. Click on **Next**.
10. Select the persistence type, **Container-Managed Persistence** or **Bean-Managed Persistence**—select **Container-Managed Persistence** in this case. You can now add and remove persistence fields (and any time later as well).
11. Click on **Next**.
12. Add superclasses (if necessary) and click on **Next**.
13. Add the methods (which you can also do at any time during the development phase). For each method, choose the method type and click on **Add**. Enter the names and return types of the methods and specify the parameters.
14. Click on **Finish**.

Result The J2EE Explorer in the SAP NWDS resembles Figure 6.7.

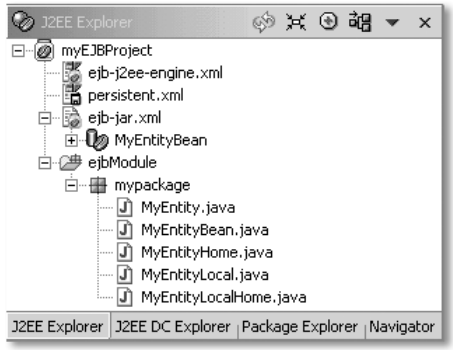


Figure 6.7 Result in the J2EE Explorer

You can now edit the entity bean in the source code and create the fields for container-managed persistence (CMP fields).

Process

1. In the J2EE Explorer window, select your project, then ejb-jar.xml, and finally the enterprise bean whose fields you want to create.
2. Select **Open** in the context menu; the bean properties appear in the right-hand window.
3. Select the **Fields** tab.
4. Select **Persistent Fields** and click **Add**. A new persistence field appears as a sub-node within the **Persistent Fields** tree structure.
5. Enter the following data in each of the fields listed below:
   - **Name**: name of the field. The SAP NetWeaver Developer Studio uses this name to create the corresponding get and set access methods. In accordance with Java conventions, the first letter of the field name is uppercase, prefaced by set or get.
   - **Fully Qualified Name**: the fully qualified name of the field type, which also must contain the package name.
   - **Array**: Choose this option to specify that the persistence field represents an array of the specified type. Enter the dimension of the array in the field that appears after you select **Array option**. You must enter values between 1 and 9.

Result

The persistence fields are now described by the corresponding *ejb-jar.xml* deployment descriptor file. Its O/R mapping was automatically written to the SAP J2EE Engine-specific deployment descriptor, *persistent.xml*. This file configures the EJB container to take over the container-managed persistence, by setting the following values and properties in the *persistent.xml* file:

- Data source and database vendor
- Type and method of lock mechanisms for the entity beans
- The O/R mapping
- The deployment properties of the finder and select SQL methods, which the container uses to optimize performance of the entity beans

During the deployment process, all the necessary code is generated by the EJB container, based on the information in the deployment descriptors.

As a result, developers no longer have to implement the access logic; they only have to declare and configure attributes and relationships.

Bean-Managed Persistence (BMP)

Bean-managed persistence is much more complicated than container-managed persistence, because you—the developer—have to program the persistence logic explicitly in the bean class.

This means you have to implement the SQL statements completely yourself. Consequently, this model makes you highly flexible in defining how the state is managed between the bean instance and the database. Entity beans that are defined through complex joins, a combination of different database systems, or other resources (such as legacy systems) generally benefit from BMP. Even though O/R mapping of the abstract schema does not meet the project's requirements, this programming model can still help.

Container-managed persistence lets you map objects to only one table, which is relatively restrictive when it comes to true distributed objects—which aren't held in just one table, but instead can be composed of multiple attributes from different data sources. Bean-managed persistence offers many more possibilities and greater flexibility in such cases.

The disadvantage of BMP is that a lot more work is required to define the beans. You have to understand the database structure and develop the logic to generate, update, and remove the data associated with an entity. You have to be very careful when dealing with the generic bean methods, especially `ejbLoad()` and `ejbStore()`. You even have to develop the search methods defined in the bean's home interface, along with the mapping of the bean attributes to the database, explicitly and manually.

A bean-managed application is not as database-independent as a container-managed entity, but is better suited to dealing with complex data.

You can use both pure vendor-specific JDBC and the native form, as well as Open SQL (SQLJ) to ensure the greatest possible database independence.

Even so, SAP recommends always using CMP (or JDO, which are described below) within object-related developments. Accordingly, we will not describe BMP in further detail here. If you would like to learn more, see the specific Java or J2EE literature.

Java Data Objects (JDO)

Java Data Objects are SAP's second recommended method for implementing object/relational persistence.

The JDO standard is a very promising technology for implementing persistent Java objects. Although JDO is one of the many Java standards and is usually mentioned in the context of J2EE, it is not included in the J2EE 1.3 or 1.4 specifications. Java Data Objects is implemented in the SAP Web AS, however, due to the many advantages it offers compared to the EJB concept.

While EJB entity beans are based on the component model of the J2EE architecture, the JDO standard tries to keep as close as possible to the Java object model. JDO lets you make almost any Java class persistent directly, independently of the architecture layer where the objects are contained. Accordingly, JDO does not require the container model of the J2EE environment, but instead adds persistence directly to the Java language.

JDO also enables you to access nearly any type of data store—relational databases, object/relational databases, or even file-based formats.

JDO is based on a byte-code transformation of the classes to be made persistent. An XML-based mapping file has to be created for each class that you want to save persistently. This file describes how the class attributes are mapped to the database tables, similar to the CMP approach. The *byte-code enhancer* then overwrites the access methods of the class and replaces them with the required SQL statements.

JDO features a Persistence Manager, which manages the life cycle, transactions, requests, and identities of the persistence objects. The query language is JDOQL *(Java Data Objects Query Language)*.

JDO can be combined with JSPs, session beans, and entity beans (BMP)—but not with container-managed entity beans (CMP)—within a J2EE environment.

Tutorial: JDO Development Cycle

Objective For the developer, developing a persistent object is an extremely structured process, with a defined procedure. The JDO development tools are not yet integrated in the SAP NetWeaver Developer Studio, however, which means all the steps have to be performed manually. This tutorial describes the development cycle for an employee object, implemented by the class `Resource`:

Process

1. Define the database tables.
2. Create the pure Java classes that you want to make persistent.

   The classes that will implement the persistence must be created and implemented in the SAP NetWeaver Developer Studio, just like any other Java classes. Ultimately, each of these classes defines objects that will be saved in a database and can be retrieved from the database.

   To create the classes, choose **New • Java Class**. Enter "Resource" as the class name.

   The new Java files open automatically and you can enter the coding. The following coding excerpt illustrates an example of such a class; it is limited to three attributes of the resource object at this point.

```
public class Resource {

   // Class attributes: the persistent fields
   //                   of a resource.
   // Also defined inside the file Resource.jdo
      private int resId;
      private String resType;
      private String description;

   // Required: a no-args constructor
   public Resource() {
     this.resId = 1;
     this.resType = "INITIAL";
     this.description = "INITIAL";
   }

   // Constructor where the ID is set
```

```
    public Resource(int resId) {
      this.resId = resId;
      this.resType = "INITIAL";
      this.description = "INITIAL";
    }

    // Implement the getter methods of the class
    public int getResId() {
      return resId;
    }

    public String getResType() {
      return resType;
    }

    public String getDescription() {
      return description;
    }

    // Implement the setter methods of the class
    public void setResType(String type) {
      resType = type;
    }

    public void setDescription(String desc) {
      description = desc;
    }
}
```

3. Define the object identity.

   JDO provides for an identity class for each persistent object, which ensures that an individual JDO instance is associated with a persistence manager that represents a specific data store object—the database.

   While the JDO standard describes three types of identity—application, data store, and nondurable identity—SAP's JDO implementation supports only the application identity. In this form, the application manages the JDO identity and holds it in the data store. In most cases, the instance identity is the primary key.

   To implement the identity, you have to create a special object identity class for each persistent class, which is defined as `static public`

`inner class ID` of the corresponding persistence class. The object identity class has a corresponding `public instance` field for each primary key field, with the same name and data type.

The object identity class has a constructor without arguments, like the persistent class. It also has a string constructor, which returns an instance as a string (like a `toString()` method).

It also has to implement a `hashCode()` method, which returns the primary key, and an `equals()` method, which uses a Boolean return value to check the instance or check which instance a given object belongs to.

The following coding illustrates the identity class for the resource object as an example:

```
import com.sap.jdo.SAPJDOHelper;
static public class Id {

// public field corresponding to the primary key of the
// PC class
public int resId;

static {
    // establish the relation: Resource$Id
    // class is the identity class for the
    // PC class Resource.
    SAPJDOHelper.registerPCClass(Resource.class);
}

public Id() {
 //required: a no-args constructor
}

public Id(int resId) {
      this.resId = resId;
}

public Id(String string) {
// required: a string constructor
// defined as the counterpart of toString()
              resId = Integer.parseInt(string);
}
```

```
public int hashCode() {
// required: implement hashCode()
            return resId;
}

public String toString() {
// required: toString() defined
// as the counterpart of the string constructor
            return Integer.toString(resId);
}

public boolean equals(Object that) {
// required: define equals()
            if (that == null || !(that instanceof Id))
                  return false;
            else
                  return resId == ((Id) that).resId;
}
}
```

4. Define the JDO metadata.

   You can now define the XML metadata for the JDO objects. To do so, each persistence class is assigned a corresponding **.jdo* file, which must be located in the same directory.

   - Choose **New · File**.
   - Enter the directory where you want to save the file and enter the same file name as the corresponding Java class file, but with extension *.jdo*.
   - You can now enter the definitions. The following listing shows the coding for the preceding Java class as an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="temp.persistence.gettingstarted.jdo">
<class name="Resource"
           identity-type="application"
           objectid-class="Resource$Id">
      <field name="resId"
```

```
            persistence-modifier="persistent"
            primary-key="true"/>
      <field name="resType"
            persistence-modifier="persistent"/>
      <field name="description"
            persistence-modifier="persistent"/>
    </class>
  </package>
</jdo>
```

5. Define the O/R mapping for the persistence classes.

   You can now create the mapping, using another XML file that you also save under the same name in the same folder, but with file extension **.map*.

   The format of this class is defined by the JDO mapping metadata Document Type Definition (DTD).

   You create this file just like the **.jdo* file; the mapping for this example is demonstrated below:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map version="1.0">
  <package name="temp.persistence.gettingstarted.jdo">
    <class name="Resource">
      <field name="resId">
        <column name="RESID" table="ZRM_RES_MASTER"/>
      </field>
      <field name="resType">
        <column name="RESTYPE" table="ZRM_RES_MASTER"/>
      </field>
      <field name="description">
        <column name="DESCRIPTION"
                table="ZRM_RES_MASTER"/>
      </field>
    </class>
  </package>
</map>
```

6. Use the JDO Enhancer to compile the code.

   Now that you have created the appropriate classes and descriptors, use the JDO Enhancer to create the classes that will communicate with the application business logic.

   Because the JDO Enhancer and the validation tools are not integrated in the Developer Studio yet, you have to perform these steps manually. To do so, we recommend using the ANT build tool, which is available as a plug-in for the Developer Studio.

   - Create a new file for your project—using the context menu—with name *build.xml*, and save it in the main directory of your project.
   - Enter the following coding (continuing the above example):

```
<project name="GettingStartedWithJDO" default="enhance"
         basedir="..">

  <property name ="sourceproject.dir"
            value="GettingStartedJDOWeb"/>
  <property name ="dictproject.dir"
            value="GettingStartedPersistenceDic"/>
  <property name ="src.dir"
            value="${sourceproject.dir}/source"/>
  <property name ="bin.dir"
            value="${sourceproject.dir}/bin"/>
  <property name ="catalog.dir"
            value="${dictproject.dir}/gen_ddic
                   /dbtables/"/>

  <property name ="enhancer"
            value="com.sap.jdo.enhancer.Main"/>

  <property name ="utility"
            value="com.sap.jdo.sql.util.JDO"/>
  <property name ="tssap"
            value="C:/Program Files/SAP/JDT/eclipse
                   /plugins"/>
  <property name ="jdo"
            value="${tssap}/com.sap.ide.eclipse.ext
                   .libs.jdo/lib/jdo.jar"/>
  <property name ="xml"
            value="${tssap}/com.tssap.sap.libs
```

```
                    .xmltoolkit
                    /lib/sapxmltoolkit.jar"/>
  <property name ="jdoutil"
            value="${tssap}/com.sap.jdo.utils
                    /lib/sapjdoutil.jar"/>
  <property name ="dictionary"
            value="${tssap}/com.sap.dictionary.database
                    /lib/jddi.jar"/>
  <property name ="logging"
            value="${tssap}/com.tssap.sap.libs.logging
                    /lib/logging.jar"/>
  <property name ="catalogreader"
            value="${tssap}/com.sap.opensql
                    /lib/opensqlapi.jar"/>
  <property name ="classpath"
            value="${jdo};${jdoutil};${xml}"/>

<property name ="classpath.check"
          value="${classpath};${dictionary};${logging};
                 ${catalogreader};${bin.dir}"/>
<target name="enhance">
  <antcall target="enhance.Resource"/>
</target>

<target name="check">
  <antcall target="check.Resource"/>
</target>

<target name="enhance.Resource">
  <java
    fork       ="yes"
    failonerror="yes"
    classname  ="${enhancer}"
    classpath  ="${classpath}">
    <arg line ="-f -d" />
        <arg value="${bin.dir}"/>
        <arg value="${src.dir}/temp/persistence/
             gettingstarted/jdo/Resource.jdo"/>
        <arg value="${bin.dir}/temp/persistence/
             gettingstarted/jdo/Resource.class"/>
      </java>
```

```
    </target>

    <target name="check.Resource">
      <java
        fork       ="yes"
        failonerror="yes"
        classname  ="${utility}"
        classpath  ="${classpath.check}">
        <arg line  ="-v -p" />
        <arg value ="${sourceproject.dir}
                     /checker.properties"
         />
        <arg value ="-c"/>
        <arg value ="${catalog.dir}"/>
        <arg value ="check"/>
        <arg value ="temp/persistence/gettingstarted/jdo
                /Resource.class"/>
    </java>
  </target>
</project>
```

- Now create a new file called *checker.properties*, again from the context menu of the project, and save it in the main folder for the project as well. Enter the following coding in this new file:

```
com.sap.jdo.sql.mapping.useCatalog=true
com.sap.jdo.sql.mapping.checkConsistency=true
com.sap.jdo.sql.mapping.checkConsistencyDeep=true
```

7. In the Java perspective of the Developer Studio, open the context menu for the *build.xml* file and choose **Run ANT...**
8. If they are not set already, set the **enhance** and **check** objects in the **Targets** tab.
9. Choose **Apply** and then **Run**. The result of the process is output in the Developer Studio console.
10. The generated classes now implement interface `javax.jdo.spi.PersistenceCapable`, which can be used by the business logic.

Figure 6.8 summarizes the development process of JDO persistence. **Result**

Although the JDO development process is quite structured, the fact that the corresponding tools for using JDO are not integrated in the Developer Studio yet is a distinct disadvantage. Therefore, you have to spend a lot of time on tasks that the appropriate tools could perform automatically. Errors are likely in larger projects because developers can quickly lose track of the manually created files.

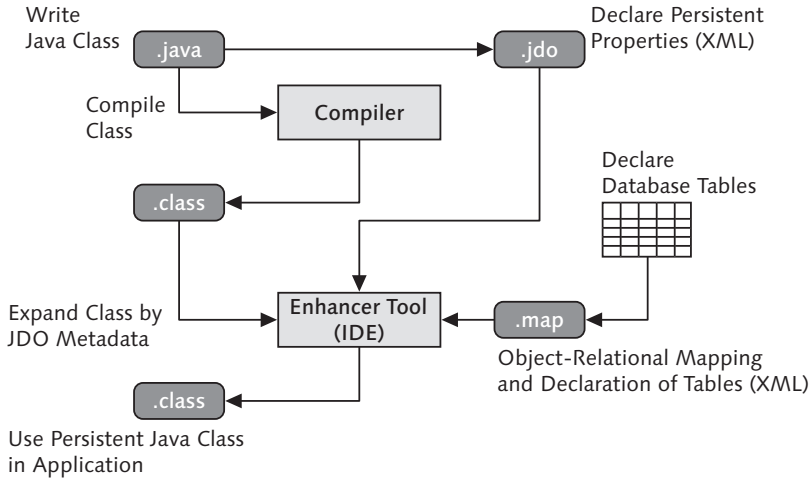


Figure 6.8 JDO Development Cycle

You probably won't be able to leverage the full potential of the JDO approach in SAP environments until SAP adds the necessary extensions to the Developer Studio.

As we already mentioned, SAP recommends using CMP or JDO for object-oriented persistence, instead of using relational-persistence by implementing raw SQL via JDBC bean-managed persistence, which is completely ignored at this time. The following table compares and contrasts the two SAP-recommended approaches and suggests possible uses for each:

JDO 1.0	Entity Beans (CMP) 2.0
Not part of the J2EE standard, that is, it is not supported by all J2EE application servers.	Part of the J2EE standard; the J2EE specifications guarantee support for CMP. CMP features a complete programming model.
Does not support direct remote calls, but they can be implemented using façade session beans.	Although direct remote calls are possible, SAP recommends using entity beans only locally, and delegating remote communication to façade session beans.
No implicit security features are available.	The CMP specifications contain security features.
JDO is available in both managed environments and open Java environments.	Only executable in managed environments such as EJB containers.
Supports inheritance and interfaces, multiple mapping, and mapping to legacy tables.	Does not support inheritance or interfaces. A bean can only be mapped to a single table.
A separate class is required for the primary keys; this implementation is not automated and is therefore time-intensive.	No primary key class is required.
The inter-object relationships are not managed centrally.	The inter-object relationships are managed centrally.
JDOQL permits dynamic queries, which always return objects. JDOQL has a Java-like syntax, however, and is not as powerful.	EJBQL can execute only static queries that return unchangeable datasets. EJBQL has a syntax comparable to OQL and is more powerful than JDOQL.
Faster development cycles.	The deployment processes are very complex and hence, much slower.
Java skills are sufficient to create new implementations.	Using EJBs requires extensive knowledge of object orientation and distributed concepts.

Table 6.2 Comparison Between JDO and Entity Beans

6.2 Middleware: Connectivity Between Applications

Direct access to the ABAP Personality

As we have already mentioned several times, all connectivity between the ABAP and Java personalities has to occur at application level. It is usually not possible for a Java application to access the ABAP instance's database

Index

C

D

K

L

M

N

O

P

Q

R

S

T

U

V

W

X