Foreword 9

1 Introduction 11

2 Technology Overview 17

2.1 Enterprise Services Architecture .......................................................... 18
2.2 SAP NetWeaver ...................................................................................... 19
  2.2.1 People Integration ........................................................................... 20
  2.2.2 Information Integration ................................................................. 20
  2.2.3 Process Integration ......................................................................... 20
  2.2.4 Application Integration ................................................................. 20
  2.2.5 Life Cycle Management ................................................................. 21
  2.2.6 Composite Application Framework ............................................. 21
2.3 System Architecture of the SAP Web Application Server .................. 21
  2.3.1 Presentation Layer ......................................................................... 23
  2.3.2 Application Layer .......................................................................... 24
  2.3.3 Database Layer ............................................................................... 24
2.4 Major Components of the SAP Web Application Server .................... 24
  2.4.1 The Internet Communication Manager (ICM) ............................... 24
  2.4.2 The ABAP Runtime Environment ............................................... 26
  2.4.3 The J2EE Engine ........................................................................... 26
  2.4.4 The Integration Engine .................................................................. 28
2.5 Database Integration ............................................................................... 30
  2.5.1 Database Independence ............................................................... 30
  2.5.2 Client Capability ........................................................................... 31
  2.5.3 Caching and Trace Mechanisms .................................................. 31
  2.5.4 Transaction Capability .................................................................. 31
  2.5.5 Object-Relational Mapping ......................................................... 32
  2.5.6 Native SQL ................................................................................... 32
2.6 Web Services ........................................................................................ 33
  2.6.1 What Are Web Services? .............................................................. 33
  2.6.2 The Web Service Paradigm ........................................................... 33
  2.6.3 The SAP Web Application Server as Web Service Client .......... 35
  2.6.4 The SAP Web Application Server as Web Service Provider ....... 35
  2.6.5 Outlook ........................................................................................ 36
2.7 Frontends .............................................................................................. 36
  2.7.1 SAP GUI for Windows ................................................................. 36
  2.7.2 SAP GUI for Java ........................................................................... 38
2.7.3 SAP GUI for HTML .......................................................... 40
2.7.4 Pure Browser Interface (BSP, JSP) ............................. 42
2.7.5 Web Dynpro ................................................................. 44

2.8 Authorization System ...................................................... 46
   2.8.1 Authorizations in the ABAP Personality .................. 46
   2.8.2 Authorizations in the Java Personality .................... 48
   2.8.3 Security of J2EE Applications ............................... 51

2.9 Versioning and Transport System .................................. 54
   2.9.1 Versioning ............................................................... 54
   2.9.2 Transport System .................................................... 56
   2.9.3 Versioning and Transport Process under Java ........ 58

2.10 Availability, Performance, Scalability ............................ 59

2.11 Integration Options for ABAP and J2EE ......................... 60
   2.11.1 J2EE Calls ABAP .................................................. 60
   2.11.2 ABAP Calls J2EE .................................................. 61

3 The "ResMan" Example Project ................................. 63
   3.1 Prerequisites ............................................................ 63
      3.1.1 Business Benefit ................................................ 63
      3.1.2 Functional Prerequisites ..................................... 64
      3.1.3 Technical Prerequisites ..................................... 65
   3.2 The Data Model ........................................................ 65
   3.3 Technical Implementation of the Prerequisites .............. 69

4 The Programming Languages of the SAP Web Application Server 71
   4.1 ABAP and ABAP Objects .......................................... 71
      4.1.1 Typical Activities .............................................. 71
      4.1.2 Basic Terminology and Concepts ......................... 72
      4.1.3 Variables and Data Types .................................. 74
      4.1.4 The Most Important Commands and Language Constructs 78
   4.2 Java ........................................................................ 90
      4.2.1 Basic Terminology and Concepts ......................... 90
      4.2.2 Object-Oriented Programming ............................. 91
      4.2.3 A First Example Program in Java ....................... 93
      4.2.4 Comments .......................................................... 94
      4.2.5 Identifiers and Keywords .................................... 95
      4.2.6 Data Types, Variables, and Constants .................. 96
      4.2.7 Operators .......................................................... 101
      4.2.8 Control Structures ............................................ 107
      4.2.9 Exception Handling ........................................... 112
      4.2.10 Methods .......................................................... 114
      4.2.11 Classes and Objects .......................................... 115
<table>
<thead>
<tr>
<th>7</th>
<th>Application Design</th>
<th>421</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>A Typical Problem</td>
<td>421</td>
</tr>
<tr>
<td>7.2</td>
<td>Design Patterns</td>
<td>422</td>
</tr>
<tr>
<td>7.2.1</td>
<td>MVC</td>
<td>422</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Façade</td>
<td>428</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Adapters</td>
<td>430</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Driver Model</td>
<td>431</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Lazy Initialization</td>
<td>434</td>
</tr>
<tr>
<td>7.3</td>
<td>Developing an ABAP Web Application</td>
<td>436</td>
</tr>
<tr>
<td>7.4</td>
<td>Developing a J2EE Web Application</td>
<td>438</td>
</tr>
<tr>
<td>8</td>
<td>Performance Aspects</td>
<td>441</td>
</tr>
<tr>
<td>8.1</td>
<td>Performance under ABAP</td>
<td>441</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Rules for Boosting Performance</td>
<td>442</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Performance Analysis Tools</td>
<td>443</td>
</tr>
<tr>
<td>8.2</td>
<td>Performance under Java</td>
<td>445</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Rules for Boosting Performance</td>
<td>447</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Performance Analysis Tools</td>
<td>450</td>
</tr>
<tr>
<td>9</td>
<td>Outlook</td>
<td>455</td>
</tr>
<tr>
<td>A</td>
<td>Glossary</td>
<td>459</td>
</tr>
<tr>
<td>B</td>
<td>Sources and Further Reading</td>
<td>479</td>
</tr>
<tr>
<td>C</td>
<td>About the Authors</td>
<td>481</td>
</tr>
<tr>
<td>Index</td>
<td>485</td>
<td></td>
</tr>
</tbody>
</table>
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 sev-
eral 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.

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 pro-
  grams that use a given logical database.

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 ser-
vices, 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.

```plaintext
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**

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-
specific 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.

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](image)

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 not 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.
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.

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.

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.

```java
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.
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.

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.

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.

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

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>'')');
```

Retrieval Logic and Persistence
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:

```java
String res_id = "1";
ssql [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.

```java
String[] values = new String[5];
MyClass ref = new MyClass();
i = 3;
ssql [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:

```java
ssql [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 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.

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 }
```

Database Connection Context

Connection Context with Data Source

Retrieval Logic and Persistence
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 then 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.
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.

The SQLJ Checker is integrated seamlessly in SAP NetWeaver Developer
Studio. When the SQLJ files are converted—which is performed automatic-
ically as soon as the data is saved—the checker runs through the embed-
ded 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 Dictio-
nary 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 .gdbtable file that contains the offline catalog. Therefore, you must
associate this file with your project:

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 .gdbtable 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 navi-
gate directly to the relevant part of the SQLJ source file, as well as set
breakpoints in it.

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. Oth-
erwise, 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.

To activate SQLJ 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 SQLJ 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 under-
lying JDBC connection, which means the former is not closed.

In the following coding example, an SQLJ connection context, ctx, is cre-
at by the JDBC connection conn. ctx and conn now share the same 
database connection. The INSERT and DELETE statements are both per-
formed for this connection and share the same transaction.

```java
#sql context MyCtx;

//...

Connection conn = ...;
Statement stmt = conn.createStatement();
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 SQL

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 per-
formed for this connection and share the same transaction.

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

//...

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

```
VALUES (100, 1, 'R', 'Conference room first floor', null, '0000100100')
}

Connection conn = ctx.getConnection();
Statement stmt = conn.createStatement();
stmt.executeUpdate(
    " DELETE FROM ZRM_RES_MASTER WHERE RESID = 1");
while (namIter.next()) {
    System.out.println(namIter.name());
}

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

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.

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.

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.

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

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.

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

Table 6.1 Rules for Mapping CMP to JDBC
Relationships are implemented as references between primary key columns and foreign key columns.

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.

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.

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.

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

- 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.

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.

**Requirements**

**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](image)

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

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.

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 (SQU) 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.

```java
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:

```java
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 exten-
sion .jdo.

You can now enter the definitions. The following listing shows the
coding for the preceding Java class as an example:

```xml
<?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"
```
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
<?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):

```xml
<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="${tssap}/com.tssap.sap.libs"
```
<property name="jdoutil" value="${tssap}/com.sap.jdoutil./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>
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:

```properties
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.
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:
Direct access to the ABAP Personality

6.2 Middleware: Connectivity Between Applications

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 directly. However, there are two main approaches to achieve this connectivity:

- **JDO 1.0**
  - Not part of the J2EE standard, that is, it is not supported by all J2EE application servers.
  - Does not support direct remote calls, but they can be implemented using façade session beans.
  - No implicit security features are available.
  - Supports inheritance and interfaces, multiple mapping, and mapping to legacy tables.
  - A separate class is required for the primary keys; this implementation is not automated and is therefore time-intensive.
  - The inter-object relationships are not managed centrally.
  - JDOQL permits dynamic queries, which always return objects. JDOQL has a Java-like syntax, however, and is not as powerful.
  - Faster development cycles.
  - Java skills are sufficient to create new implementations.

- **Entity Beans (CMP) 2.0**
  - Part of the J2EE standard; the J2EE specifications guarantee support for CMP.
  - CMP features a complete programming model.
  - Although direct remote calls are possible, SAP recommends using entity beans only locally, and delegating remote communication to façade session beans.
  - The CMP specifications contain security features.
  - Only executable in managed environments such as EJB containers.
  - Does not support inheritance or interfaces. A bean can only be mapped to a single table.
  - No primary key class is required.
  - The inter-object relationships are managed centrally.
  - EJBQL can execute only static queries that return unchangeable datasets. EJBQL has a syntax comparable to OQL and is more powerful than JDOQL.
  - The deployment processes are very complex and hence, much slower.
  - Using EJBs requires extensive knowledge of object orientation and distributed concepts.

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

Table 6.2 Comparison Between JDO and Entity Beans
Index

3-tier architecture 22

A
ABAP 71
  business logic 345
  presentation logic 362
ABAP application 436
ABAP development environment 138
ABAP Dictionary 139, 141, 459
ABAP Editor 139, 459
ABAP Interpreter 459
ABAP Objects 83, 137, 147, 459
ABAP Personality 26, 459
  authorizations 46
ABAP program 459
ABAP VM (Virtual Machine) 26, 459
ABAP Web service 315
  SOAP request 327
  SOAP response 328
ABAP Workbench 138, 207, 459
ABAP/Java integration 60
Abstract class 459
Abstract method 459
abstract, keyword, Java 129
ActiveX 459
Adapters 430
Adaptive RFC 396
Aggregation 459
Alert monitor 452
ALV 138
American Standard Code for Information Interchange (ASCII) 460
Applet 460
Applet container 172, 460
Applets 49, 168
Application class 150
Application client archive 173
Application client container 460
Application hierarchy 459
Application integration 20
Application layer 24
Application Link Enabling (ALE) 29, 315, 459
Application Modeler 402
Application platform 20
Application programming interface (API) 460
Application server 460
Application Tracing Service 451
Application tracing, Java 451
Arrays, Java 100, 121
  multidimensional 100
Assembler 445
Assembly project 58
Association 460
Attributes, Java 118, 460
  hiding 126
Authentication
  J2EE 52
  Java 203
Authorization 46, 460
  EJBs 53
  J2EE 53
  Java 204
Authorization concept 460
Authorization field 46
Authorization object 46
Authorization profile 47
Authorization system 46
AWT 460
Back-end 460
BAPI 345, 460
BAPI Explorer 139, 346, 460
Basic authentication 53
Batch input 460
Batch input session 460
Bean-managed persistence (BMP) 179, 286
Beans 461
Binding, JNDI 190, 461
Bottom-up modularization 461
BPM 461
Branching, CVS 245
Break, statement, Java 110
Breakpoint 145, 461
BSP application 149, 437
  stateful 152
  stateless 153
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSP extensions</td>
<td>155, 364, 367, 438</td>
</tr>
<tr>
<td>Buffer control, JSP</td>
<td>187</td>
</tr>
<tr>
<td>Build</td>
<td>461</td>
</tr>
<tr>
<td>Business Application Programming Interfaces (BAPI)</td>
<td>137, 345</td>
</tr>
<tr>
<td>Business Connector</td>
<td>28, 461</td>
</tr>
<tr>
<td>Business Framework</td>
<td>461</td>
</tr>
<tr>
<td>Business intelligence</td>
<td>20</td>
</tr>
<tr>
<td>Business logic</td>
<td>344, 461</td>
</tr>
<tr>
<td>Java</td>
<td>170, 214</td>
</tr>
<tr>
<td>Business object</td>
<td>346, 461</td>
</tr>
<tr>
<td>Business Object Builder</td>
<td>139, 461</td>
</tr>
<tr>
<td>Business Object Repository</td>
<td>346</td>
</tr>
<tr>
<td>Business Server Pages</td>
<td>23, 42, 138, 151, 364, 461</td>
</tr>
<tr>
<td>Business Workflow</td>
<td>461</td>
</tr>
<tr>
<td>Byte code</td>
<td>461</td>
</tr>
<tr>
<td>Byte code enhancer</td>
<td>287</td>
</tr>
</tbody>
</table>

**C**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>445</td>
</tr>
<tr>
<td>Caching</td>
<td>31, 461</td>
</tr>
<tr>
<td>CALL METHOD, statement</td>
<td>86</td>
</tr>
<tr>
<td>Call-by-reference</td>
<td>461</td>
</tr>
<tr>
<td>Call-by-value</td>
<td>461</td>
</tr>
<tr>
<td>CASE, statement</td>
<td>79</td>
</tr>
<tr>
<td>Cast, Java</td>
<td>99</td>
</tr>
<tr>
<td>Casting</td>
<td>461</td>
</tr>
<tr>
<td>CATT</td>
<td>461</td>
</tr>
<tr>
<td>CBS</td>
<td>461</td>
</tr>
<tr>
<td>Certificate interfaces</td>
<td>202</td>
</tr>
<tr>
<td>CGI</td>
<td>462</td>
</tr>
<tr>
<td>Change Management Service (CMS)</td>
<td>58, 239</td>
</tr>
<tr>
<td>Change request</td>
<td>462</td>
</tr>
<tr>
<td>Changing parameter</td>
<td>462</td>
</tr>
<tr>
<td>Class</td>
<td>459, 462</td>
</tr>
<tr>
<td>ABAP</td>
<td>147</td>
</tr>
<tr>
<td>ABAP Objects</td>
<td></td>
</tr>
<tr>
<td>abstract</td>
<td>89</td>
</tr>
<tr>
<td>final</td>
<td>89</td>
</tr>
<tr>
<td>abstract</td>
<td>462</td>
</tr>
<tr>
<td>concrete</td>
<td>462</td>
</tr>
<tr>
<td>inner</td>
<td>462</td>
</tr>
<tr>
<td>Java</td>
<td>115</td>
</tr>
<tr>
<td>abstract</td>
<td>129</td>
</tr>
<tr>
<td>defining</td>
<td>116</td>
</tr>
<tr>
<td>extending</td>
<td>125</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS ... ENDCLASS, statement</td>
<td>84</td>
</tr>
<tr>
<td>Class attribute</td>
<td>462</td>
</tr>
<tr>
<td>Class Browser</td>
<td>462</td>
</tr>
<tr>
<td>Class Builder</td>
<td>139, 147, 462</td>
</tr>
<tr>
<td>Class inheritance</td>
<td>462</td>
</tr>
<tr>
<td>Class library</td>
<td>462</td>
</tr>
<tr>
<td>Class members</td>
<td>119</td>
</tr>
<tr>
<td>Class method</td>
<td>462</td>
</tr>
<tr>
<td>Class pools</td>
<td>73</td>
</tr>
<tr>
<td>Client</td>
<td>462</td>
</tr>
<tr>
<td>Client tier</td>
<td>52</td>
</tr>
<tr>
<td>Clients, Java</td>
<td>167</td>
</tr>
<tr>
<td>CMS</td>
<td>462</td>
</tr>
<tr>
<td>Code Inspector</td>
<td>144, 443, 445</td>
</tr>
<tr>
<td>Collision control, CVS</td>
<td>245</td>
</tr>
<tr>
<td>COM</td>
<td>462</td>
</tr>
<tr>
<td>COM/DCOM Connector</td>
<td>29</td>
</tr>
<tr>
<td>Comments, Java</td>
<td>94</td>
</tr>
<tr>
<td>Common Client Interface (CCI)</td>
<td>195</td>
</tr>
<tr>
<td>Compiler</td>
<td>462</td>
</tr>
<tr>
<td>Component Build Service (CBS)</td>
<td>58, 215, 238, 461</td>
</tr>
<tr>
<td>Component controller</td>
<td>399, 462</td>
</tr>
<tr>
<td>Component interface</td>
<td>398, 462</td>
</tr>
<tr>
<td>Component model technologies, Java</td>
<td>174</td>
</tr>
<tr>
<td>Component, Web Dynpro</td>
<td>462</td>
</tr>
<tr>
<td>Composite Application Framework</td>
<td>21, 462</td>
</tr>
<tr>
<td>Composite profile</td>
<td>47</td>
</tr>
<tr>
<td>Composite role</td>
<td>47</td>
</tr>
<tr>
<td>Composition</td>
<td>463</td>
</tr>
<tr>
<td>Compression</td>
<td>427</td>
</tr>
<tr>
<td>Computing Center Management System (CCMS)</td>
<td>59, 450</td>
</tr>
<tr>
<td>Concurrent Version System (CVS)</td>
<td>244</td>
</tr>
<tr>
<td>Connection context</td>
<td>273</td>
</tr>
<tr>
<td>Connection management, Java</td>
<td>195</td>
</tr>
<tr>
<td>Connection pool</td>
<td>264, 300, 304</td>
</tr>
<tr>
<td>Connectors</td>
<td>28</td>
</tr>
<tr>
<td>Consolidation routes</td>
<td>159, 463</td>
</tr>
<tr>
<td>Consolidation system</td>
<td>463</td>
</tr>
<tr>
<td>Constants</td>
<td></td>
</tr>
<tr>
<td>ABAP</td>
<td>74</td>
</tr>
<tr>
<td>Java</td>
<td>99</td>
</tr>
<tr>
<td>Constructors</td>
<td>117, 463</td>
</tr>
<tr>
<td>Container</td>
<td>463</td>
</tr>
<tr>
<td>Container services</td>
<td>171</td>
</tr>
</tbody>
</table>
Container-managed persistence (CMP) 179, 280, 463
Containers, Java 170
Context (Web Dynpro) 463
Context Builder 463
Context Editor 404
Continue, statement, Java 111
Contracts 195
Control Framework 362, 463
Control structures
ABAP 78
Java 107
Control technology 137
Controller Editor 404
Controllers 150, 398, 409, 423
ABAP 438
Java 439
CORBA 33, 178, 190, 463
Coverage Analyzer 443, 445
Credentials 203
Cryptographic services 202
CSF 463
CSS 463
CSV 463
Custom controller 397, 399, 416
Custom controls 362
Custom Tags 463
Customizing requests 161
CVS 240

Data Control Language (DCL) 255
Data Definition Language (DDL) 255
Data Dictionary project, Java creating 226
Data element 464
Data encapsulation 464
Data Manipulation Language (DML) 255
Data Modeler 139, 141, 463
Data objects, ABAP 74
Data structures, Java deployment 233
Data types 141, 464
ABAP 75
Java creating 228
JCo 302
Database 463
Database commit 464
Database independence 30
Database integration 30
Database layer 24
Database queries performance 442
Database rollback 464
Database tables 142
defining 230
Database transaction 464
Database, logical 464
DB access layer 265
DCOM 33, 464
DCOM Connector 315
Debugger 464
Debugging 144
Declarative programming 464
Decrement operators 101
DELETE, statement 83
Delivery routes 160
Delphi 445
Delta handling 426
deltav 243
Deploy process 233
Deployment 440, 464
Deployment descriptor 182, 464
Design patterns 464
Design Time Repository (DTR) 58, 215, 238
Development class 464
Development cycle, Web Dynpro 465
Development environment 465
Development objects
ABAP 143
Java 216
Development paradigm, Java 212
Development projects
ABAP 441
Development system 465
Dialog program 465
Diff algorithm 245
Distributed statistics record (DSR) 59, 450
DNS 190
DO, statement 80, 109
Doc comment 132
Document management systems 240
Document Object Model (DOM) 196
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Type Definitions (DTD)</td>
<td>196</td>
</tr>
<tr>
<td>Documentation conventions</td>
<td>132</td>
</tr>
<tr>
<td>DOM</td>
<td>465</td>
</tr>
<tr>
<td>Domain</td>
<td>143</td>
</tr>
<tr>
<td>Driver model</td>
<td>431</td>
</tr>
<tr>
<td>DTD</td>
<td>465</td>
</tr>
<tr>
<td>DTR</td>
<td>465</td>
</tr>
<tr>
<td>Dynpro</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>362</td>
</tr>
<tr>
<td></td>
<td>391</td>
</tr>
<tr>
<td></td>
<td>465</td>
</tr>
<tr>
<td>E</td>
<td>465</td>
</tr>
<tr>
<td>EAR</td>
<td>465</td>
</tr>
<tr>
<td>ebXML</td>
<td>465</td>
</tr>
<tr>
<td>ebXML Registry</td>
<td>197</td>
</tr>
<tr>
<td>eCATT</td>
<td>165</td>
</tr>
<tr>
<td>Eclipse</td>
<td>465</td>
</tr>
<tr>
<td>EIS</td>
<td>465</td>
</tr>
<tr>
<td>EJB</td>
<td>465</td>
</tr>
<tr>
<td>assembly project</td>
<td>221</td>
</tr>
<tr>
<td>class diagram</td>
<td>332</td>
</tr>
<tr>
<td>container</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>350</td>
</tr>
<tr>
<td></td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>465</td>
</tr>
<tr>
<td>project</td>
<td>220</td>
</tr>
<tr>
<td>proxy class</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>312</td>
</tr>
<tr>
<td>Electronic Data Interchange (EDI)</td>
<td>29</td>
</tr>
<tr>
<td>Encapsulation, Java</td>
<td>128</td>
</tr>
<tr>
<td>Endless loops, Java</td>
<td>110</td>
</tr>
<tr>
<td>Enjoy controls</td>
<td>465</td>
</tr>
<tr>
<td>enjoySAP</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>465</td>
</tr>
<tr>
<td>Enterprise application project</td>
<td>222</td>
</tr>
<tr>
<td></td>
<td>372</td>
</tr>
<tr>
<td></td>
<td>387</td>
</tr>
<tr>
<td>Enterprise archives</td>
<td>173</td>
</tr>
<tr>
<td>Enterprise information system (EIS)</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td>170</td>
</tr>
<tr>
<td>Enterprise JavaBeans</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>176</td>
</tr>
<tr>
<td>deployment</td>
<td>182</td>
</tr>
<tr>
<td>Enterprise JavaBeans archives</td>
<td>173</td>
</tr>
<tr>
<td>Enterprise Services Architecture (ESA)</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>36</td>
</tr>
<tr>
<td>Entity bean</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>279</td>
</tr>
<tr>
<td></td>
<td>466</td>
</tr>
<tr>
<td>life cycles</td>
<td>177</td>
</tr>
<tr>
<td>equals(), method, Java</td>
<td>122</td>
</tr>
<tr>
<td>ERP</td>
<td>466</td>
</tr>
<tr>
<td>Error handling</td>
<td>466</td>
</tr>
<tr>
<td>JSP</td>
<td>186</td>
</tr>
<tr>
<td>ESA</td>
<td>466</td>
</tr>
<tr>
<td>Escape sequences, Java</td>
<td>97</td>
</tr>
<tr>
<td>Event</td>
<td>466</td>
</tr>
<tr>
<td>Event block</td>
<td>466</td>
</tr>
<tr>
<td>Event handler</td>
<td>152</td>
</tr>
<tr>
<td>Event, ABAP Objects</td>
<td>89</td>
</tr>
<tr>
<td>Exception</td>
<td>466</td>
</tr>
<tr>
<td>Exception handling</td>
<td>466</td>
</tr>
<tr>
<td>Java</td>
<td>112</td>
</tr>
<tr>
<td>Exchange connections</td>
<td>276</td>
</tr>
<tr>
<td>Executable programs, ABAP</td>
<td>72</td>
</tr>
<tr>
<td>Export parameter</td>
<td>466</td>
</tr>
<tr>
<td>Extended program check</td>
<td>144</td>
</tr>
<tr>
<td>extends, keyword, Java</td>
<td>125</td>
</tr>
<tr>
<td>Extreme Programming (XP)</td>
<td>249</td>
</tr>
<tr>
<td>Façade pattern</td>
<td>428</td>
</tr>
<tr>
<td>Field symbols</td>
<td>77</td>
</tr>
<tr>
<td>Filters</td>
<td>423</td>
</tr>
<tr>
<td>servlets</td>
<td>219</td>
</tr>
<tr>
<td>final, keyword, Java</td>
<td>127</td>
</tr>
<tr>
<td>for, statement, Java</td>
<td>109</td>
</tr>
<tr>
<td>Form-based authentication</td>
<td>53</td>
</tr>
<tr>
<td>Framework</td>
<td>466</td>
</tr>
<tr>
<td>Function</td>
<td>466</td>
</tr>
<tr>
<td>Function Builder</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>466</td>
</tr>
<tr>
<td>Function code</td>
<td>466</td>
</tr>
<tr>
<td>Function group</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>466</td>
</tr>
<tr>
<td>Function library</td>
<td>466</td>
</tr>
<tr>
<td>Function module</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>466</td>
</tr>
<tr>
<td>Garbage collection, Java</td>
<td>122</td>
</tr>
<tr>
<td>Garbage collector</td>
<td>466</td>
</tr>
<tr>
<td>Generation limit</td>
<td>145</td>
</tr>
<tr>
<td>Get method</td>
<td>118</td>
</tr>
<tr>
<td>Group, J2EE</td>
<td>52</td>
</tr>
<tr>
<td>GUI status</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>466</td>
</tr>
<tr>
<td>GUI title</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>466</td>
</tr>
<tr>
<td>GuiXT</td>
<td>466</td>
</tr>
<tr>
<td>Hash</td>
<td>466</td>
</tr>
<tr>
<td>Host expressions</td>
<td>272</td>
</tr>
<tr>
<td>Host variables, Java</td>
<td>271</td>
</tr>
<tr>
<td>Hprof</td>
<td>447</td>
</tr>
<tr>
<td>HTMLB</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>438</td>
</tr>
<tr>
<td></td>
<td>467</td>
</tr>
<tr>
<td>HTTP</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>241</td>
</tr>
<tr>
<td>HTTP error pages</td>
<td>186</td>
</tr>
<tr>
<td>Index</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>IBM WebSphere</td>
<td>19</td>
</tr>
<tr>
<td>IAC</td>
<td>467</td>
</tr>
<tr>
<td>ICM</td>
<td>467</td>
</tr>
<tr>
<td>IDE</td>
<td>467</td>
</tr>
<tr>
<td>Identifiers, Java</td>
<td>95</td>
</tr>
<tr>
<td>IDoc</td>
<td>29, 315, 467</td>
</tr>
<tr>
<td>IF, statement, ABAP</td>
<td>78</td>
</tr>
<tr>
<td>IF, statement, Java</td>
<td>107</td>
</tr>
<tr>
<td>implements, keyword, Java</td>
<td>130</td>
</tr>
<tr>
<td>Import parameter</td>
<td>467</td>
</tr>
<tr>
<td>Import queue</td>
<td>162</td>
</tr>
<tr>
<td>import, statement, Java</td>
<td>124</td>
</tr>
<tr>
<td>Inbound call</td>
<td>300</td>
</tr>
<tr>
<td>Inbound plug (Web Dynpro)</td>
<td>392, 467</td>
</tr>
<tr>
<td>Include programs</td>
<td>73</td>
</tr>
<tr>
<td>Includes</td>
<td>467</td>
</tr>
<tr>
<td>JSP</td>
<td>189</td>
</tr>
<tr>
<td>Increment operators</td>
<td>101</td>
</tr>
<tr>
<td>Information integration</td>
<td>20</td>
</tr>
<tr>
<td>Inheritance</td>
<td>88</td>
</tr>
<tr>
<td>ABAP Objects</td>
<td>124, 127</td>
</tr>
<tr>
<td>preventing</td>
<td>127</td>
</tr>
<tr>
<td>Initialization</td>
<td>467</td>
</tr>
<tr>
<td>INSERT, statement</td>
<td>81</td>
</tr>
<tr>
<td>Instance</td>
<td>467</td>
</tr>
<tr>
<td>Instance members</td>
<td>119</td>
</tr>
<tr>
<td>Instantiation</td>
<td>467</td>
</tr>
<tr>
<td>Integration Engine</td>
<td>24, 30</td>
</tr>
<tr>
<td>Interface</td>
<td>467</td>
</tr>
<tr>
<td>Interface inheritance</td>
<td>467</td>
</tr>
<tr>
<td>Interface pools</td>
<td>73</td>
</tr>
<tr>
<td>Interface, ABAP</td>
<td>147</td>
</tr>
<tr>
<td>Interfaces, ABAP Objects</td>
<td>87</td>
</tr>
<tr>
<td>Interfaces, Java</td>
<td>129</td>
</tr>
<tr>
<td>Intermediate code</td>
<td>467</td>
</tr>
<tr>
<td>Intermediate Language (IL)</td>
<td>26</td>
</tr>
<tr>
<td>Internal table</td>
<td>77, 467</td>
</tr>
<tr>
<td>Internet Application Component (IAC)</td>
<td>40, 137</td>
</tr>
<tr>
<td>Internet Communication Framework</td>
<td>152</td>
</tr>
<tr>
<td>Internet Communication Manager (ICM)</td>
<td>23, 24</td>
</tr>
<tr>
<td>Internet Inter-Object Request Broker Protocol (IIOP)</td>
<td>178</td>
</tr>
<tr>
<td>Internet service, ABAP</td>
<td>148</td>
</tr>
<tr>
<td>Internet Transaction Server (ITS)</td>
<td>18, 137, 390, 467</td>
</tr>
<tr>
<td>Interpreter</td>
<td>467</td>
</tr>
<tr>
<td>J</td>
<td>91, 467</td>
</tr>
<tr>
<td>J2EE</td>
<td>security 51</td>
</tr>
<tr>
<td>J2EE applications, architecture</td>
<td>167</td>
</tr>
<tr>
<td>J2EE component</td>
<td>468</td>
</tr>
<tr>
<td>J2EE module</td>
<td>468</td>
</tr>
<tr>
<td>J2EE server</td>
<td>171, 468</td>
</tr>
<tr>
<td>J2ME</td>
<td>91, 468</td>
</tr>
<tr>
<td>J2SE</td>
<td>91, 468</td>
</tr>
<tr>
<td>Jakarta Struts</td>
<td>424, 439, 440</td>
</tr>
<tr>
<td>JAR</td>
<td>173, 222, 468</td>
</tr>
<tr>
<td>JarClientAPI</td>
<td>222</td>
</tr>
<tr>
<td>Java</td>
<td>468</td>
</tr>
<tr>
<td>business logic</td>
<td>347</td>
</tr>
<tr>
<td>presentation logic</td>
<td>369</td>
</tr>
<tr>
<td>security</td>
<td>48</td>
</tr>
<tr>
<td>versioning and transport</td>
<td>58</td>
</tr>
<tr>
<td>Java API for XML Registration Services (JAXR)</td>
<td>197</td>
</tr>
<tr>
<td>Java applet</td>
<td>468</td>
</tr>
<tr>
<td>Java application</td>
<td>438, 468</td>
</tr>
<tr>
<td>Java Application Descriptor</td>
<td>459</td>
</tr>
<tr>
<td>Java archive</td>
<td>222</td>
</tr>
<tr>
<td>Java Authentication and Authorization Service (JAAS)</td>
<td>202</td>
</tr>
<tr>
<td>Java byte code</td>
<td>468</td>
</tr>
<tr>
<td>Java Community Process</td>
<td>451</td>
</tr>
<tr>
<td>Java Connector (JCo)</td>
<td>28, 298</td>
</tr>
<tr>
<td>Java Cryptography Architecture (JCA)</td>
<td>202</td>
</tr>
<tr>
<td>Java Cryptography Extensions (JCE)</td>
<td>202</td>
</tr>
<tr>
<td>Java Data Dictionary</td>
<td>269</td>
</tr>
<tr>
<td>Java Data Objects (JDO)</td>
<td>198, 287</td>
</tr>
<tr>
<td>Java Database Connectivity</td>
<td>198</td>
</tr>
<tr>
<td>Java Database Objects (JDO)</td>
<td>217</td>
</tr>
<tr>
<td>Java Development Infrastructure (JDI)</td>
<td>58, 237</td>
</tr>
<tr>
<td>Java Dictionary</td>
<td>214, 217, 223, 407, 468</td>
</tr>
<tr>
<td>Java IDL</td>
<td>468</td>
</tr>
<tr>
<td>Java Message Service (JMS)</td>
<td>191, 354, 355</td>
</tr>
<tr>
<td>Java Naming and Directory Interface (JNDI)</td>
<td>173, 175, 190, 359</td>
</tr>
</tbody>
</table>
Java Native Interface (JNI) 298
Java platform 91
Java Remote Method Protocol (JRMP) 178
Java security model 51
Java Server Pages (JSP) 23, 42, 169, 184, 219, 370
error handling 186
life cycle 186
technology 468
Java Standard TagLib (JSTL) 185
Java Transaction API (JTA) 192, 200
Java Virtual Machine (JVM) 90
Java Virtual Machine Profiler Interface (JVMPI) 447
Java Web services 331
Java Web Start 468
Java XML API (JAXP) 196
Java XML RPC API 197
Java, concepts 90
JavaBeans 174, 176, 468
components 188
deployment 182
JavaMail 468
JavaOS 468
JavaScript 468
JavaServer Faces 424, 439, 440
JAXM 468
JAXP 468
JAXR API 197
JBC 469
JCA 469
JCo 60, 61, 469
JCP 469
JDBC 170, 261, 264, 265, 469
API 197
drivers 199
result set 278
with SQLJ 276
JDI 469
JDK 469
JDO 469
development 288
JDOM 469
JES 469
JFC 469
Jini 469
JMS 469
client 354
provider 354
JMX 451, 469
JNDI 469
lookup 439
lookup service 171
JNI 469
JNLP 469
JRA 470
JRE 470
JUnit 248
JVM 470
JWS 470
K
Key management interfaces 203
Keywords, Java 95
Knowledge Management (KM) 20
L
Layout manager 470
Lazy initialization 434
LDAP 190, 470
Life cycle management 21
List 470
Listeners, servlets 219
Lists 363
Literals, ABAP 74
Local object 470
Lock object 143, 470
Logging, Java 450
Logical databases 256
Loops, ABAP 79
LUW 470
M
main, method, Java 120
Mapping 470
Mass transport 162
Master Data Management (MDM) 20
ME, self reference, ABAP 117
Media types 187
Members 116
Memory management 445
Memory pipes 25
Menu Painter 145, 470
Message broker 355
Message management, Java 196
Message-driven bean 354, 470
deployment descriptor 359
life cycle 357
Metadata 470
Method 471
ABAP
redefining 126
ABAP Objects 85
abstract 471
Java 114, 118
abstract 129
overwriting 126
METHOD ... ENDMETHOD, statement 85
Method signature 114
Microsoft .NET 19
Middleware 297, 471
MIDlet 471
MIDP 471
MIME 471
MIME objects 151
MIME Repository 156
Model 394, 408, 423
ABAP 437
Java 439
Modification Browser 139, 471
Modifiers 128, 131
MODIFY, statement 82
Module 471
Module pool 72, 471
Monitoring functions
Java 450
Multilingual capability 425
Multiple output formats 426
Multithreading 471
MVC 42, 138, 370, 422, 471
authentication 423
authorization 423
data visualization 425
hierarchy and distribution 424
multilingual capability 425
validation 423

N
Naming conventions
ABAP 132
Java 131
Native JDBC 262
Native SQL 32, 265, 471
.NET 459
.NET Connector 29
Next screen 471
null, keyword 101

O
O/R mapping 281
Object 471
Object catalog 471
Object class 46
Object list 471
Object Navigator 140, 471
Object services 257
Object/relational persistence 279
Object-oriented programming 91
ABAP 83
attributes 92
class definition 92
classes 92
encapsulation 93
inheritance 93
methods 92
objects 92
Object-relational mapping 32, 472
Objects, Java 116, 121
creating 118
finalization 122
ODBC 472
OLE 472
Open Database Connectivity (ODBC) 198
Open SQL 30, 80, 255, 472
Performance 441
Open SQL Engine 217, 263, 265, 268
Open SQL for Java 261
Operators, Java
+ 105
assignment 104
bit 102
Boolean 104
instanceof 105
new 105
point operator (.) 105
priorities 106
relational 103
type conversion 105
Outbound call 306
Outbound plug (Web Dynpro) 392, 472
Outline view 404
Overloading 115, 472
Overwriting 126
preventing 127

P
Packages 143, 472
Java 123
Packaging, Java 172
PAI 362, 472
Parameters 472
ABAP 75
Java 115
Parent class 472
Patterns 472
PBO 362, 472
People integration 20
Performance
ABAP 441
analysis tools 443
Java 445
Performance trace 145
Java 451
Persistence Service 257
Persistent objects 257
Personality 472
Pluggable Authentication Module (PAM) 203
POH 362
Pointer arithmetic 445
Point-to-point model, JMS 193
Polymorphism 472
POV 362
PreparedStatement object 266
Presentation layer 23
Presentation logic 361
Pretty Printer 472
Private 472
Process integration 20
Processing block 473
Production system 473
Profilers 59, 447
Program status 473
Program structure, ABAP 72
Program type 473
Programmatic interface 398
Programs 144
Project life cycle, Java 207
Properties view 404
protected 473
public 473
Publish/subscribe model, JMS 194

Q
Queue 356, 357

R
R/3 17
RAR 173, 473
Realm 52
Reference 473
Reference equivalence 122
Remote Function Call 29
Report 144
Report program 473
Repository 473
Resource adapter archives 173
Return, statement, Java 111
Reuse components 138, 473
Reuse Library 139, 473
Revisions, CVS 245
RFC 29, 60, 61, 298, 473
Rich client 473
RMI 190, 473
Role, J2EE 47, 52
Runtime analysis 140, 145, 443, 473
Runtime environment 473

S
SAAJ API 197
SAML 473
Sandbox model 49
SAP Business Warehouse (BW) 20
SAP Enterprise Portal 20
SAP Exchange Infrastructure (XI) 20, 197, 315
SAP GUI 473
SAP GUI for HTML 40, 213
SAP GUI for Java 38
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAP GUI for Windows</td>
</tr>
<tr>
<td>SAP J2EE Engine</td>
</tr>
<tr>
<td>standards</td>
</tr>
<tr>
<td>SAP List Viewer</td>
</tr>
<tr>
<td>SAP Mobile Infrastructure</td>
</tr>
<tr>
<td>SAP NetWeaver</td>
</tr>
<tr>
<td>SAP NetWeaver Developer Studio</td>
</tr>
<tr>
<td>Java Dictionary</td>
</tr>
<tr>
<td>perspectives</td>
</tr>
<tr>
<td>testing</td>
</tr>
<tr>
<td>user interface</td>
</tr>
<tr>
<td>SAP Query</td>
</tr>
<tr>
<td>SAP technology</td>
</tr>
<tr>
<td>SAP Web Application Server</td>
</tr>
<tr>
<td>architecture</td>
</tr>
<tr>
<td>as Web service client</td>
</tr>
<tr>
<td>as Web service provider</td>
</tr>
<tr>
<td>components</td>
</tr>
<tr>
<td>frontends</td>
</tr>
<tr>
<td>transport system</td>
</tr>
<tr>
<td>versioning</td>
</tr>
<tr>
<td>SAP xEM</td>
</tr>
<tr>
<td>SAP xMA</td>
</tr>
<tr>
<td>SAP xPD</td>
</tr>
<tr>
<td>SAP xRPM</td>
</tr>
<tr>
<td>SAPconnect</td>
</tr>
<tr>
<td>SAPPone</td>
</tr>
<tr>
<td>SAX</td>
</tr>
<tr>
<td>Screen</td>
</tr>
<tr>
<td>Screen Painter</td>
</tr>
<tr>
<td>Scriptlets, Java</td>
</tr>
<tr>
<td>SDM</td>
</tr>
<tr>
<td>Search help</td>
</tr>
<tr>
<td>Security</td>
</tr>
<tr>
<td>Security concept</td>
</tr>
<tr>
<td>Security management, Java</td>
</tr>
<tr>
<td>Security policy, Java</td>
</tr>
<tr>
<td>SELECT, statement</td>
</tr>
<tr>
<td>Selection screen</td>
</tr>
<tr>
<td>Selection table</td>
</tr>
<tr>
<td>Server cache</td>
</tr>
<tr>
<td>Service</td>
</tr>
<tr>
<td>Service gateway</td>
</tr>
<tr>
<td>Service interface</td>
</tr>
<tr>
<td>Service Provider Interface (SPI)</td>
</tr>
<tr>
<td>Service-oriented architecture (SOA)</td>
</tr>
<tr>
<td>Servlet</td>
</tr>
<tr>
<td>Servlet specification</td>
</tr>
<tr>
<td>Session</td>
</tr>
<tr>
<td>Session beans</td>
</tr>
<tr>
<td>as Web services</td>
</tr>
<tr>
<td>Sessions, Java servlets</td>
</tr>
<tr>
<td>Set method, Java</td>
</tr>
<tr>
<td>Short-circuit evaluation</td>
</tr>
<tr>
<td>Signature</td>
</tr>
<tr>
<td>Simple API for XML (SAX)</td>
</tr>
<tr>
<td>Simple Object Access Protocol (SOAP)</td>
</tr>
<tr>
<td>Single transport</td>
</tr>
<tr>
<td>SL</td>
</tr>
<tr>
<td>SOAP</td>
</tr>
<tr>
<td>SOAP Runtime</td>
</tr>
<tr>
<td>SOAP transport binding</td>
</tr>
<tr>
<td>SOAP with Attachments</td>
</tr>
<tr>
<td>SOAP with Attachments API for Java (SAAJ)</td>
</tr>
<tr>
<td>Software Delivery Manager (SDM)</td>
</tr>
<tr>
<td>Software Logistics (SL)</td>
</tr>
<tr>
<td>SQL</td>
</tr>
<tr>
<td>SQL for Java (SQLJ)</td>
</tr>
<tr>
<td>SQL processor</td>
</tr>
<tr>
<td>SQL processor layer</td>
</tr>
<tr>
<td>SQL Studio, creating data</td>
</tr>
<tr>
<td>SQL trace</td>
</tr>
<tr>
<td>SQLJ</td>
</tr>
<tr>
<td>debugging</td>
</tr>
<tr>
<td>development</td>
</tr>
<tr>
<td>Syntax</td>
</tr>
<tr>
<td>SQLJ Checker</td>
</tr>
<tr>
<td>SQLJ result set iterator</td>
</tr>
<tr>
<td>Stateful session beans</td>
</tr>
<tr>
<td>Stateless session beans</td>
</tr>
<tr>
<td>Statement cache</td>
</tr>
<tr>
<td>Statement pooling</td>
</tr>
<tr>
<td>static, keyword, Java</td>
</tr>
<tr>
<td>Status</td>
</tr>
<tr>
<td>Structure</td>
</tr>
<tr>
<td>Stub</td>
</tr>
<tr>
<td>Stub/skeleton</td>
</tr>
<tr>
<td>Subclass</td>
</tr>
<tr>
<td>Subroutine pools</td>
</tr>
<tr>
<td>Subroutines</td>
</tr>
<tr>
<td>SunONE</td>
</tr>
</tbody>
</table>
super, keyword, Java 125
Superclass 475
Swing 475
Switch, statement, Java 108
Synchronized, statement, Java 111
Syntax check 144, 475
Syntax elements, ABAP 72
System data container 165
System field 475

T
Table, internal 476
Tag 476
Tag Browser 367
Tag library 367, 476
Tagging, CVS 246
TagLibs 371
TagLibs, JSP 185
Team collaboration, Java 237
Template 476
Test configuration 166
Test data container 166
Test scripts 165
Testing
ABAP 165
Java 248
Text elements 476
Text symbols 476
Thin client 476
this, keyword, Java 117
Threads, Java 111, 476
Three-tier architecture 476
Throw, statement, Java 112
throws, statement, Java 113
Title bar 476
Top-down modularization 476
Topic 355, 356
Total cost of ownership (TCO) 19
Tracing 31
Transaction 476
SCI 443, 445
SCOF 445
SCOV 443
SE11 459
SE24 462
SE30 443
SE37 466
SE81 459
ST05 443, 444
Transaction capability 31
Transaction code 476
Transaction data 476
Transaction management, Java 195, 196
Transaction Service 257
Transport group 476
Transport layer 159, 160, 476
Transport Management System (TMS) 159, 164
Transport Organizer 56, 139, 163, 477
extended view 164
tools 164
Transport process 57
Transport protocols, EJB 177
Transport request 477
Transport system 56, 157
Transporting 476
Try-catch-finally, statements, Java 113
Type conversions, Java 99

U
UDDI 33, 34, 197, 477
UI element 477
UML 477
UML-XMI 395
Unicode 477
UPDATE, statement 82
URL mappings 386
User master record 47
User, J2EE 52
Using parameter 477
UTF-8 477

V
Value domain 477
Variables
ABAP 74
Java 98
Variants 166
Vendor SQL 264
Version catalog, ABAP 157
Version control, CVS 245
Versioning 54
ABAP 157
Java 237
View 152, 400, 410, 423, 477