Spend More Time Modeling Your Java Applications and Less Time Trying to Persist Your Data — Java Data Objects (JDO) Makes It Easy!

Markus Küfer and Zornitsa Yankulova

Markus Küfer,
Developer, SAP AG

Zornitsa Yankulova,
Information Developer,
SAP Labs Bulgaria

(complete bios appear on page 102)

As the power and complexity of Java applications increase to serve expanding business needs, so does the need for the reliable availability — the "persistence" — of the data in your data stores. In the SAP world, persisting data usually requires manually writing SQL statements in your application code via Java Database Connectivity (JDBC) or SQL Java (SQLJ) that directly access the database tables in the data store to retrieve, update, and store data. But who wants to be bothered with endlessly writing, and rewriting, the SQL code in your application every time a table element, or for that matter your data store, changes? Wouldn't it be great if you could persist data without having to change your application code, and without having to know any language other than Java?

With SAP Web Application Server (SAP Web AS) 6.40, you can. SAP Web AS 6.40 includes the standard Java persistence technologies defined in Java Specification Request 12, version 1.0.1 — Enterprise JavaBeans entity beans with container-managed persistence (EJB CMP) and Java Data Objects (JDO). With these APIs, you manipulate persistent objects that represent the data in the data store, rather than accessing the database directly. Synchronization between the persistent object and the data it represents happens transparently behind the scenes, where the required SQL statements and JDBC connections are generated automatically for you. The SAP implementations of EJB CMP and JDO are also built on the Open SQL Engine, which is part of the J2EE Engine in SAP Web AS. This means that Open SQL features like caching, portability, scalability, and diagnostics, and J2EE capabilities like transaction management and security, are available as well.

While EJB CMP is the API of choice for distributed, component-

based J2EE application architectures,[1] not all Java applications are built using EJB components, which is a powerful, but complex framework that can carry a lot of overhead. JDO is based on the object model of the Java programming language rather than the J2EE component model, and can be used to persistence-enable virtually any Java class. It is a more lightweight approach that is easier to implement, since you need know little more than Java and a few JDO basics, and it can be used in any Java application, as well as J2EE.

This article introduces you to the JDO API and how it works. We will show you how to create persistent objects using JDO, and we will walk you through some real-world coding examples that show you how to take advantage of JDO in your own business applications (the complete code listings for these examples is available for download at **www.SAPpro.com**; you might find it useful to take a moment to print out the code listings to use as a reference throughout the article). This article will serve as a valuable source of information for software project managers on how JDO can save development time and costs, and increase program and system maintainability by reducing complexity. For software engineers, this article will provide a solid knowledge of basic JDO concepts, and will show you how to make effective use of the SAP JDO implementation in your own applications.

> ✓ ***Note!***
>
> *This article assumes sound Java knowledge, a good understanding of J2EE, and a grounding in object persistence basics.*[2]

[1] EJB CMP was covered in detail in the article "Persist Data for Your J2EE Applications with Less Effort Using Entity Beans with Container-Managed Persistence (EJB CMP)" (*SAP Professional Journal*, July/August 2004).

[2] For a detailed overview of persistence in SAP Web AS 6.40, see the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" (*SAP Professional Journal*, May/June 2004).

Why JDO?

JDO is essentially an enhancement to the Java programming language that provides persistence capabilities for ordinary Java objects via a simple Java class. As a consequence, all object-oriented programming techniques supported by Java can be applied to persistent Java classes — for example, a persistent class can inherit from other persistent classes, as well as from transient classes. The lightweight Java-centric view provided by JDO constitutes an elegant way to connect application code with any underlying data store. You can even use different storage paradigms, such as object databases, relational databases, or lightweight file stores for embedded systems, without having to modify your application code to accommodate them. For programmers with Java know-how, the JDO approach to persistence requires only basic knowledge about how JDO works.

The JDO API is also much simpler to learn and use than its persistence counterparts — JDBC, SQLJ, and EJB CMP. It is a simple interface that you use to implement a single, persistence-capable class. This enables application programmers to focus on modeling their application domains using object-oriented design and analysis instead of coping with relational structures and SQL statements, or adapting to complex component-based frameworks.

JDO is also designed to integrate seamlessly into J2EE environments, such as SAP Web AS, and can be used in a variety of implementation scenarios other than J2EE, providing you with flexibility and portability.

How Does JDO Work?

In its simplest definition, JDO provides a facility for transparently binding Java objects to their persistent representations in a data store. It requires neither an extensive persistence middleware or infrastructure, nor special coding techniques.

To free the application developer from providing a persistence middleware or infrastructure, JDO must

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

have the capability to detect *which* objects from the underlying database are needed, and *when* to retrieve, create, update, or delete them. Therefore, the JDO implementation has to track an object's state. This can be achieved by having the objects to be persisted implement an interface that notifies the JDO runtime about the object's state, and lets it know which attributes to access. In fact, this is the way JDO works.

When using JDO to effect persistence, a "persistible" class must implement an interface called *PersistenceCapable*. Instead of requiring the application developer to manually implement this interface, JDO provides a tool for this purpose called the JDO Enhancer, which is used to modify the class's byte code.[3] At design time, the JDO Enhancer implements the *PersistenceCapable* interface in the class behind the scenes, and replaces references to persistent attributes with generated method calls that communicate with the JDO implementation at runtime. An accompanying XML descriptor file created by the developer denotes the class as "persistence-capable," and defines which attributes of a class are persistent (more on this later). We refer to a class that implements the *PersistenceCapable* interface as a "persistence-capable" or "JDO" class in order to differentiate it from unenhanced Java classes (i.e., classes that lack the persistence capability).

Now that JDO knows *what* to persist, *how* does JDO actually persist that data in SAP environments? In compliance with SAP's overall Java Persistence Framework,[4] the SAP JDO implementation is based on Open SQL and therefore persists into relational storage. This means that there must be some kind of mapping between the classes and their persistent attributes, and the database tables and columns in the data store. Since the JDO 1.0.1 specification is data-store-independent, it does not define an automatic mapping — the mapping metadata is specific to the particular JDO implementation. For this reason, the developer has to manually provide the mapping information in an additional XML descriptor file (more on this later).

Figure 1 Basic JDO Concepts

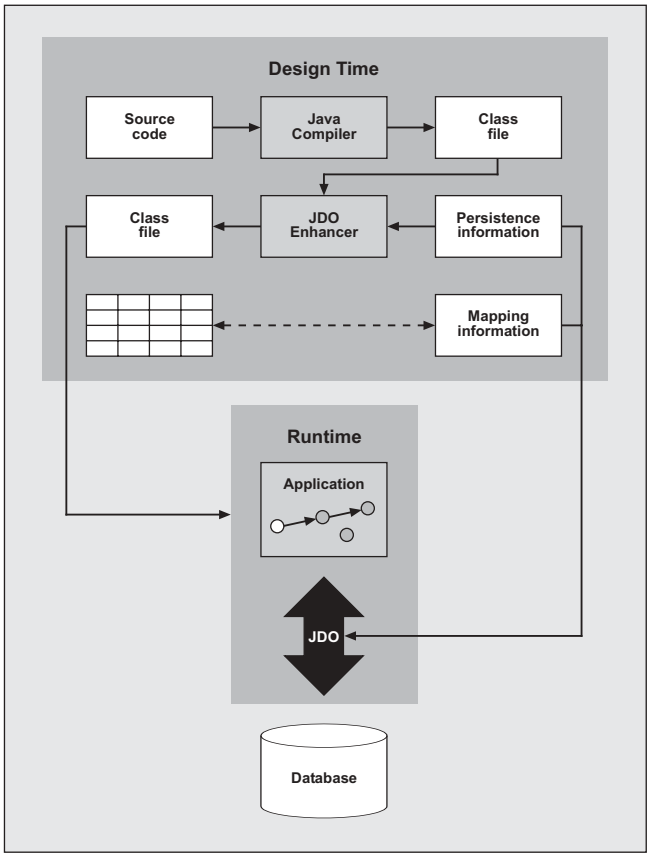


Figure 1 illustrates JDO's basic concepts. As you can see, everything starts with writing a plain old Java class that is to be persisted. Once compiled, at design time the JDO Enhancer enriches the class's byte code using the persistence information provided in an XML descriptor file. The mapping information contained in an additional XML descriptor file is also provided at design time. At runtime, the JDO implementation uses the persistence and mapping information to guarantee a synchronized, consistent state for the persistence-capable class instances in memory and on the database.

In the following sections, we will walk through how to develop persistence-capable Java classes

[3] Since byte-code enhancement is the usual way to implement the *PersistenceCapable* interface, the JDO Enhancer is included in SAP NetWeaver Developer Studio as part of SAP's JDO implementation (see the section "Automating the Enhancement of the JDO Classes").

[4] See the article "A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications" (*SAP Professional Journal*, May/June 2004).

(JDO classes) using SAP NetWeaver Developer Studio.[5] Later in the article, we will show you how to use them in a J2EE application that adheres to the JDO programming model.

Let's now look at the following tasks in particular:

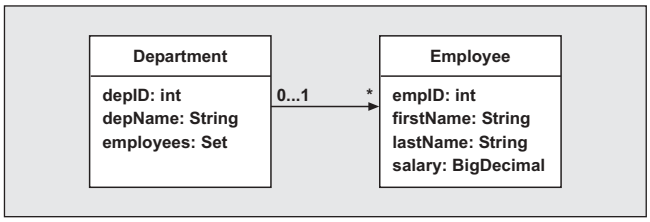
- Identifying the objects you want to persist
- Creating a session bean to serve as a façade
- Creating JDO classes and identities for the objects to be persisted
- Declaring the persistence metadata for the JDO classes
- Automating the enhancement of the JDO classes
- Mapping the persistence metadata to the data store

Identifying the Objects You Want to Persist

In order to correctly specify the object-relational mapping for the objects you want to persist, it is always a good idea to start with a UML domain model. **Figure 2** represents such a model using a UML class diagram. As you can see, we have identified two entities to persist: a Department class and an Employee class that have a uni-directional 1:*n* relationship — that is, the Department class uses data from the Employee class, but not the other way around. The Department class's attributes are *depId*, *depName*, and *employees*. The Employee class's attributes are *empId*, *firstName*, *lastName*, and *salary*. The Department class attribute *employees* is a set of the Employee class attributes.

Keep in mind that when you are transferring the UML into code, you do not have to consider persistency aspects, since this is an orthogonal concept in JDO; you simply code your Java classes as you normally do. The only thing the JDO specification

Figure 2 UML Diagram of the Department and Employee Classes



requires is that you provide a constructor taking no arguments (a "no-arg" constructor) in your class code — that's it! There is no need for bean-style getter and setter methods, although you might find them helpful to control read or write access to your data (as you will see later in the article, we found it convenient to use them in the example application). Persistent attributes can also be public, protected, or private. (By virtue of the Java language specification, attributes that are transient cannot be persisted.)

Creating a Session Bean to Serve As a Façade

Writing enterprise applications requires that the application server provide multiple capabilities:

- Security
- Communication between objects in different application domains (remoting)
- Distributed transaction management
- Availability
- Scalability
- A means to separate business logic from presentation logic

The J2EE standard provides a useful way to fulfill these requirements — stateless session beans,[6] which

[5] For a detailed introduction to using this tool to develop and deploy Java applications, including creating EJBs, JSP/servlets, and tables, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" in the May/June 2004 issue of *SAP Professional Journal*.

[6] EJBs come in three flavors: entity (fine-grained objects that represent persistent data in a data store); session (coarse-grained objects that implement behavior and processes); and message-driven (synchronous message "consumers" that integrate the EJB architecture with messaging systems like the Java Message Service).

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

offer remote calls for coarse-grained business logic methods that aggregate several local, more fine-grained method calls. Such beans represent a single client on the server side, and allow for declarative transaction demarcation using the Java Transaction API (JTA).[7]

Since JDO integrates seamlessly with J2EE, we can use a stateless session bean as a short-lived façade for our JDO classes. The session bean will then implement the procedural business logic that works with the classes.

To create the bean, first create an EJB module project (*GettingStartedJDOBean* in the example) in SAP NetWeaver Developer Studio. Then add a stateless session bean to the project called *HRInfo* as part of the package *jdoexample.businesslogic.ejb* (the system will automatically add *Bean* to the session bean's name and generate its classes and interfaces). The project name will appear in the J2EE Explorer view on the SAP NetWeaver Developer Studio screen, and will contain an *ejbModule* folder and two automatically generated deployment descriptors for the bean: a standard EJB descriptor called *ejb-jar.xml* that specifies the bean's structure, and one for SAP-specific settings called *ejb-j2ee-engine.xml*. Using the SAP NetWeaver Developer Studio XML editor, add the *HRInfoBean.java* code listed in the download at **www.SAPpro.com** to the *ejb-jar.xml* file.[8]

Creating JDO Classes and Identities for the Objects to Be Persisted

We next add two new Java classes to the EJB module project's *ejbModule* folder as part of a different package called *jdoexample.persistencelogic*, in order to separate the persistence logic from the business logic. The complete source code for the classes (*Employee.java* and *Department.java*) is included in the download available at **www.SAPpro.com**.

We must also provide an "identity" object (class) for each of the JDO classes. In the Java virtual machine (JVM), each Java object has an identity of its own that corresponds to its allocation in main memory. This object identity is subject to change over space and time, which is fine for transient objects, since they will not outlive the JVM, but not for persistent objects. JDO therefore uses a separate identity object that encapsulates the persistent object's identity. This identity object associates the in-memory Java object with its underlying data store representation. It is possible to have several instances of a persistent object allocated in the JVM at a given time. Each, although a different in-memory Java object, would have the same JDO object identity that returns *true* on calling the *equals()* operator.

But what about the JDO identity class's structure? The JDO specification generally allows for two different kinds of JDO identity: data store and application. Data store identity leaves the definition and management of the identity class to the JDO implementation. The values that make up the identity object are in no way connected to the attributes of the persistent class. In contrast, application identity requires the JDO object identity class to consist of attributes of the persistent class that uniquely identify an instance of the persistent class. Application identity is more commonly used in large-scale business applications having semantic key attributes that are part of the object model. Therefore, SAP's JDO implementation supports application identity.

The JDO specification imposes the following requirements on the implementation of a JDO identity class:

- The class must be public.
- The class must be serializable.
- The class must have a public no-arg constructor.
- There must be a public field of the same name

[7] For further discussion of JTA transactions and JDO, see the section "Implementing Transaction Management" later in the article.

[8] Creating EJBs was covered in detail in the article "Persist Data for Your J2EE Applications with Less Effort Using Entity Beans with Container-Managed Persistence (EJB CMP)" (*SAP Professional Journal*, July/August 2004).

Figure 3 Object Identity Class for the Employee JDO Class

```
Static public class Id implements Serializable {
      // public field corresponding to the primary key of the PC class
      public int empId;
      static { // establish the relation: Employee$Id class
               // is the identity class for the PC class Employee.
            SAPJDOHelper.registerPCClass(Employee.class);
      }

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

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

      public Id(String string) { // required: a string constructor
                  // defined as the counterpart of toString()
            empId = Integer.parseInt(string);
      }
      public int hashCode() { // required: implement hashCode()
            return empId;
      }

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

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

and type for each field that uniquely identifies the persistence-capable class (i.e., each field that is flagged as a *primary-key* field).

- The *equals()* and *hashCode()* methods must use the values of all the fields that uniquely identify the persistence-capable class.
- The *toString()* method must return a string representation of the object identity instance.
- The class must have a constructor that takes a string returned from the *toString()* method of an object identity instance and constructs an equivalent object identity instance.

As a developer, once you have provided this class, you will not have to deal with it unless you are interested in the identity of a JDO instance. You don't have to instantiate and populate an application identity class upon the creation of a JDO instance, for example — this happens automatically. Simply ignore the identity objects until you are searching for a particular instance of a JDO class (which we explain further in the later section "Retrieving Persistent Objects") or if you want to check two persistent objects for equality.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 4 The Department Class's Persistence Metadata File

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="jdoexample.persistencelogic">
    <class name="Department" identity-type="application" objectid-
class="Department$Id">
      <field name="depId"     persistence-modifier="persistent" primary-
key="true"/>
      <field name="name"      persistence-modifier="persistent"/>
      <field name="employees" persistence-modifier="persistent">
        <collection "element-type="jdoexample.persistencelogic.Employee"/>
      </field>
    </class>
  </package>
</jdo>
```

> ✓ ***Note!***
>
> *The JDO specification does not specify the implementation of a persistent class's equals() method in the same way it does for a JDO identity class. That's why you should use the JDO identity class's equals() method to test whether two in-memory instances represent the same persistent object.*

For the Department class, the *depId* attribute is a promising candidate for uniquely identifying department instances. For the Employee class, the *empId* attribute guarantees uniqueness as well.

For simplification, we'll implement the object identity classes as static inner classes of their corresponding JDO classes. **Figure 3** shows the static inner object identity class (*Employee$Id*) for the Employee JDO class. The definition of the Department identity class (*Department$Id*) can be found in the download available at **www.SAPpro.com**.

The static initializer code ensures that the identity object class is known by the JDO implementation. This is necessary if you instantiate the identity object without instantiating the persistent object itself.

Declaring the Persistence Metadata for the JDO Classes

At this point, the classes we have created are still just plain old Java classes. We next need to provide persistence metadata that will enable the JDO Enhancer to turn our Department and Employee classes into persistence-capable JDO classes.

Persistence metadata has to be provided in (you guessed it) an XML file with a *.jdo* extension. You must create this file manually using the XML editor in SAP NetWeaver Developer Studio and add it to the EJB module project's *ejbModule* folder as part of the *jdoexample.persistencelogic* package. The most efficient approach is to follow the same naming convention for the persistence metadata files that you followed for the *.java* class files they are associated with (*Department.java* and *Employee.java* in the example). The *Department.jdo* and the *Employee.jdo* files have to be loaded as a resource by the same class loader that loads the *Department.java* and *Employee.java* files, so it is common practice to place the *.java* files and the *.jdo* files in the same folder (in this case, *ejbModule/jdoexample/persistencelogic*).

Figure 4 shows the Department class's persistence metadata file (see the download available at **www.SAPpro.com** for the Employee class's metadata file):

- The header element specifies the XML version and encoding, along with the document type definition (DTD) location.

- The <*jdo*> element contains all the other elements that compile the class's persistence metadata:

  - The fully qualified package name (in this case, *jdoexample.persistencelogic*).

  - The object identity type (in this case *application* since that is the type supported by the SAP JDO implementation) and name (*Department$Id*).

  - The field names for each attribute of the class. (Keep in mind that if you do not provide these field names, they will be defaulted depending on several parameters, which might cause unexpected results.)

  - The persistence flag for each attribute (*persistent*, *transient*, or *none*). In Figure 4, we mark all of the attributes as *persistent*, meaning they will be persisted if the object's instance is persisted. Marking a field as *transient* will cause the attribute's value to be handled transactionally (i.e., rolling back a transaction will reset a changed value to what it was at the beginning of the transaction). An attribute flag of *none* will simply cause JDO to ignore this attribute (no transactional awareness, no persistence).

  - The *primary-key* flag for the fields that uniquely identify an object (*depID* in the example). Remember that this is the field we used for the Department class's identity object.

  - The field *employees* differs from the others — this is the field that corresponds to the set of Employee class attributes, and it therefore contains an additional element named *collection*. The *collection* element indicates that this is a field for an attribute of type *java.util.Collection*, which is used to specify the type of the elements contained in the collection. In the example, this is *jdoexample.persistencelogic.Employee*.

✓ *Default Fetch Groups*

It is also possible to define a set of class attributes that are likely to be accessed each time an instance of the class is retrieved. This allows for performance optimization in the JDO implementation — it can load or update those attributes with a single SQL statement instead of having one for each attribute. However, constantly loading attributes that are not likely to be accessed can negatively affect performance, so be sure that the attributes in such a set really are accessed frequently. You can tell the JDO implementation about a group of frequently accessed fields by setting a "default fetch group" attribute to "true" for all of those fields in the class's .jdo file. For example, the Employee class's firstName and lastName attributes could form such a default fetch group. For fields not belonging to the default fetch group (such as salary, for example) simply omit the "default fetch group" attribute as follows:

```
...
<field name="firstName"
 persistence-  modifier="persistence"
 default-fetch-group="true"/>
<field name="lastName"
 persistence-  modifier="persistent"
 default-fetch-group="true"/>
<field name="salary"
 persistence-modifier="persistent"/>
...
```

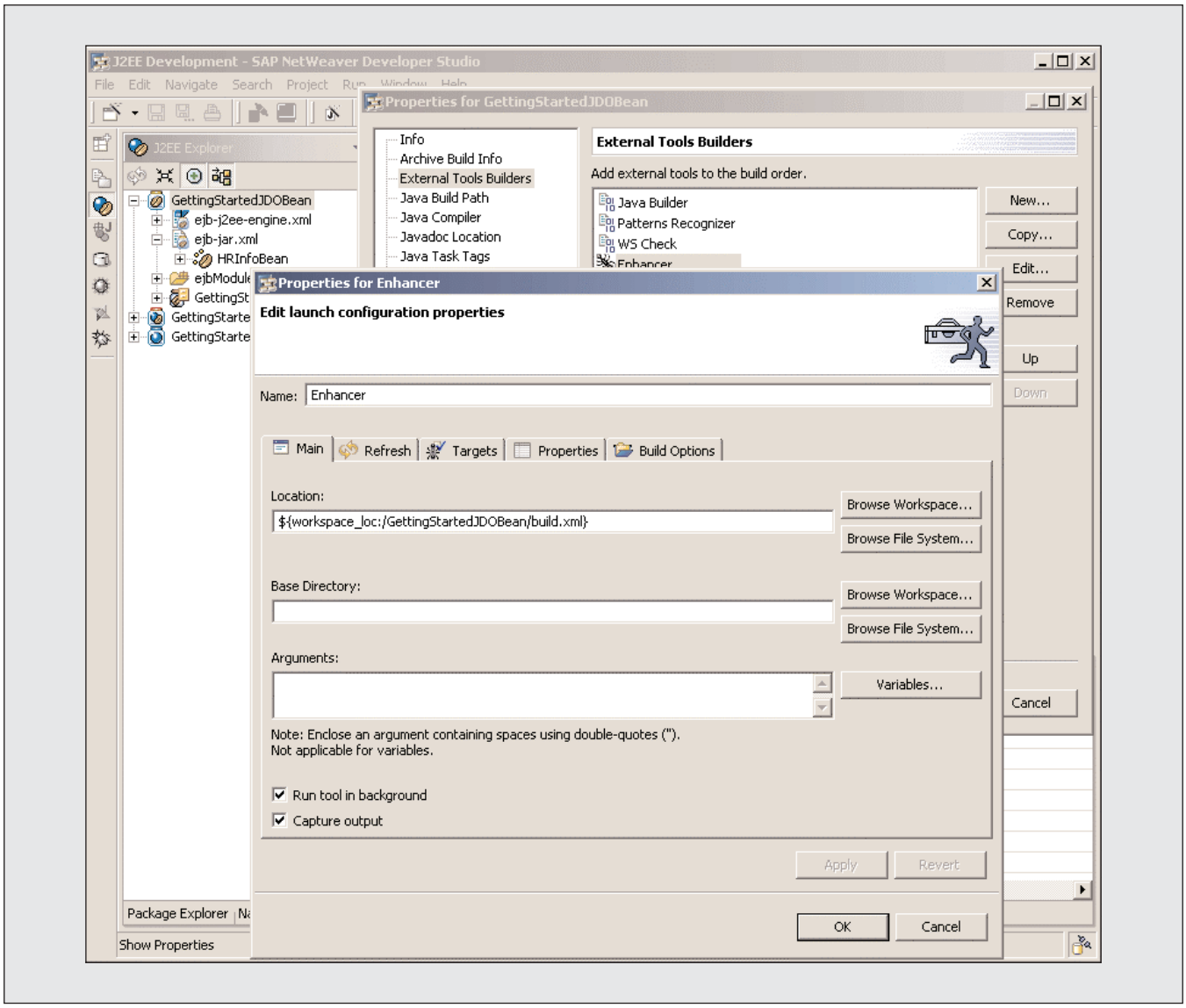
Upon retrieving an object from the database, the JDO implementation loads the default fetch group.

Automating the Enhancement of the JDO Classes

We are now ready to turn our plain Java classes into persistence-capable classes. To do this, we enhance the classes' byte code using the JDO Enhancer. There is no need to trigger the enhancer manually to enhance the class files — there is a convenient way to automate this by making use of the

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 5 Creating a Launch Configuration for the JDO Enhancer



Apache Ant[9] functionality embedded in SAP NetWeaver Developer Studio (see **Figure 5**). Simply follow these steps:

1. In the J2EE Explorer, add a *build.xml* file to the root folder of the *GettingStartedJDOBean* project. You must write this file manually using the XML editor in SAP NetWeaver Developer Studio.[10]

2. Right-click on the EJB project name (*GettingStartedJDOBean* in the example) and select *properties*.

3. In the *Properties* dialog, select *External Tools*

[9] Apache Ant is a Java-based build tool. It is similar to the Java development tool Make, but instead of relying on Unix or Windows commands for some of the configuration, which essentially ties the build environment to a platform, Apache Ant remains platform-independent by using XML-based configuration files. For more information, visit **www.ant.apache.com**.

[10] For your convenience, we provide a ready-to-use *build.xml* file in the download available at **www.SAPpro.com**. To save coding time, you can use this file as a template for your own JDO implementations.

Figure 6 The Enhancement Target for the Department Class

```
<target name="enhance.Department">
  <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/Department.jdo" />
    <arg value=
"${bin.dir}/temp/persistence/gettingstarted/jdo/Department.class" />
  </java>
</target>
```

Builders, and click on *New…*. In the pop-up that appears, select the external tool type *Ant Build*. Then name the tool in the *Properties* dialog (*Enhancer* in the example).

4. On the *Main* tab, set the *Location* to point to the *build.xml* file (*GettingStartedJDOBean/build.xml*). On the other tabs, you can customize the tool's behavior to your needs — for example, you can select the classes you want to enhance or when to start the enhancement.

Following these steps will cause the JDO Enhancer to weave the persistence metadata provided by the *Department.jdo* and *Employee.jdo* files into their respective *Department.java* and *Employee.java* class files each time they are compiled from their respective Java files.

Figure 6 is a snippet from the *build.xml* file created for the example that shows the enhancement target for the Department class. This enhancement target runs the JDO Enhancer class *com.sap.jdo.enhancer.Main* with options *-f* (for forcing an overwrite of the result file) and *-d* (for the destination folder). The last two parameters are the two input files: *Department.jdo* and *Department.class*.

Mapping the Persistence Metadata to the Data Store

Before we can finally start to work with the JDO classes we have created, we need to provide the metadata that describes how to map the persistent attributes of the classes to relational structures in the data store. The SAP JDO implementation offers a variety of mapping variants to suit all needs. Here we will focus on mapping simple attributes of the Department and Employee class, like *firstName* and *salary*, as well as the Department class's *employees* attribute, which forms a uni-directional 1:*n* relationship between the Department and the Employee classes (for more on advanced mapping features see the sidebar on the next page, and also refer to the SAP NetWeaver Developer Studio help).

When working with SAP Web AS, instead of creating tables directly on the target database, you specify them on an abstract, metadata level using SAP NetWeaver Developer Studio. Create a Dictionary project (*GettingStartedPersistenceDic* in the example), add the tables (*TMP_JDO_DEPARTMENT* and *TMP_JDO_EMPLOYEE*), and then deploy them to the concrete target server database. The JDO runtime will utilize this dictionary metadata to run on top of SAP's Open SQL relational persistence framework.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Advanced JDO Mapping Concepts

This sidebar explores two advanced JDO mapping concepts — bi-directional relationships and inheritance.

Bi-Directional Relationships

In the discussion in the article, the Department class declares a uni-directional relationship to the Employee class via the *employees* attribute of the Department class — i.e., the Department attribute *employees* references the attributes of the Employee class, but not the other way around.

Now imagine that we add an attribute *department* of type *Department* to the Employee class that references the attributes of the Department class. This would turn the relationship between the Department and Employee classes into a bi-directional relationship. Since we want JDO to handle this relationship as well, we would add the field *department* in the *Employee.jdo* file, as follows:

> ✓ ***Tip***
>
> *Remember that in Java, references are directed. Imagine that you have instantiated a department and added some employees to it. Updating one reference — e.g., setting one employee's department attribute to null — will in no way cause this employee to vanish from the department's set of employees. Remember that this holds true for persistent classes as well: JDO will not update relationships of persistent objects behind the scenes, so always keep your Java object model consistent. Otherwise, the persistent objects' database representations will be out of sync with their in-memory representations.*

```
<field name="department" persistence-modifier="persistent"/>
```

We would also extend the Employee class's field element in the *.map* file via an additional relationship field element:

```
<relationship-field name="department" multiplicity="one">
  <foreign-key name="DEPARTMENT_TO_EMPLOYEE"
   foreign-key-table="TMP_JDO_EMPLOYEE"
   primary-key-table="TMP_JDO_DEPARTMENT">
    <column-pair foreign-key-column="DEPID"
     primary-key-column="DEPID"/>
  </foreign-key>
</relationship-field>
```

As you can see, this is the description of the foreign key relationship on the database. The relationship field element's multiplicity is *one*, indicating that this describes the reference to the side whose cardinality is 1.

Inheritance

To provide you with an idea of how similar JDO is to Java, let's briefly discuss a typical object-oriented concept — inheritance. Mapping classes to tables becomes considerably more complex in the context of persistent class inheritance.

A persistent class's superclass could be transient or persistent. If it is transient, simply have the subclass

(continued on next page)

(continued from previous page)

inherit from the superclass and provide the persistence and mapping metadata for the subclass as we did for the example JDO classes in the article. When it comes to inheriting from persistent classes, keep in mind that mapping and inheritance are based on two principles:

- Mapping is inherited.
- Mapping cannot be redefined.

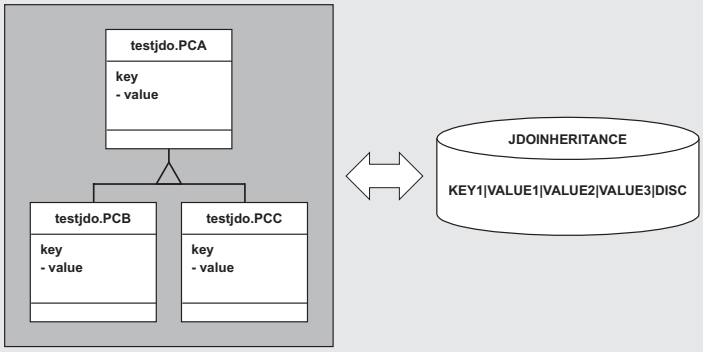
We will now consider two ways to map inheritance hierarchies to tables:

- Mapping subclasses and superclasses to one table
- Mapping each class to a separate table

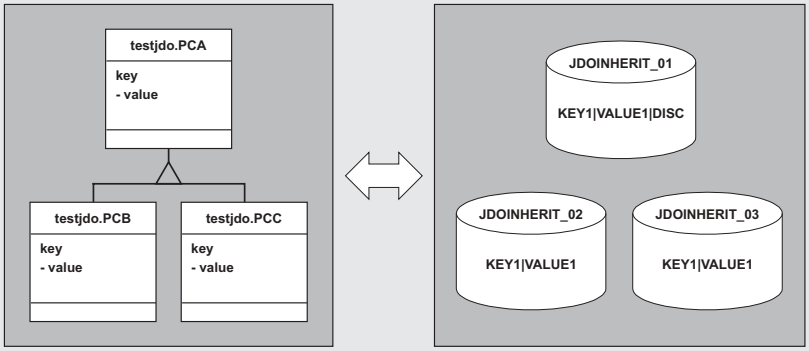
The first approach is appropriate if the class hierarchy is small and the subclasses do not have many declared fields; the second approach is appropriate if many subclasses have many declared fields.

To help you visualize the two strategies, let's look at an example. Assume subclasses PCB and PCC inherit from superclass PCA, and that all three classes define a private attribute *value*.

The first strategy maps the superclass and subclasses to one table that has columns for each class's *value* attribute. Remember that in general, a class corresponds to a single table, so we now need to differentiate between the classes. This is achieved by introducing an additional discriminator column named *DISC* that holds a class identifier (see the first diagram to the right).



The second strategy requires three tables, one for each class, so there is no need to define a discriminator column (see the second diagram to the right). When implementing this strategy, a class's attributes are mapped to several tables: some attributes are defined by the superclass and are mapped to the superclass's table, while other attributes are declared in the subclass only and map to the subclass's table. Therefore, such database entries are identified by the same primary key (i.e., the identity object key), which is mapped to more than one table.



 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 7 *Table TMP_JDO_DEPARTMENT*

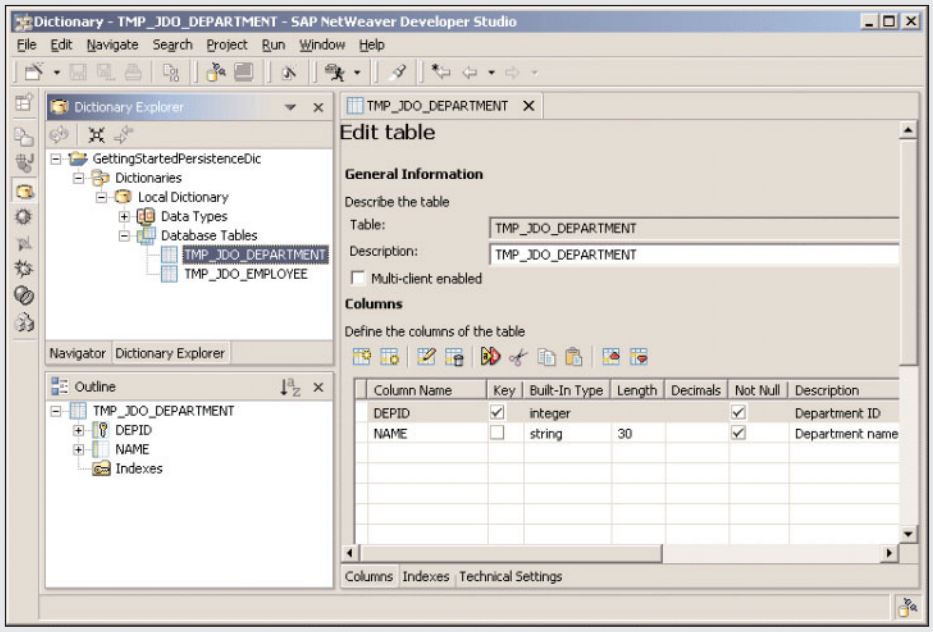


Figure 8 *Table TMP_JDO_EMPLOYEE*

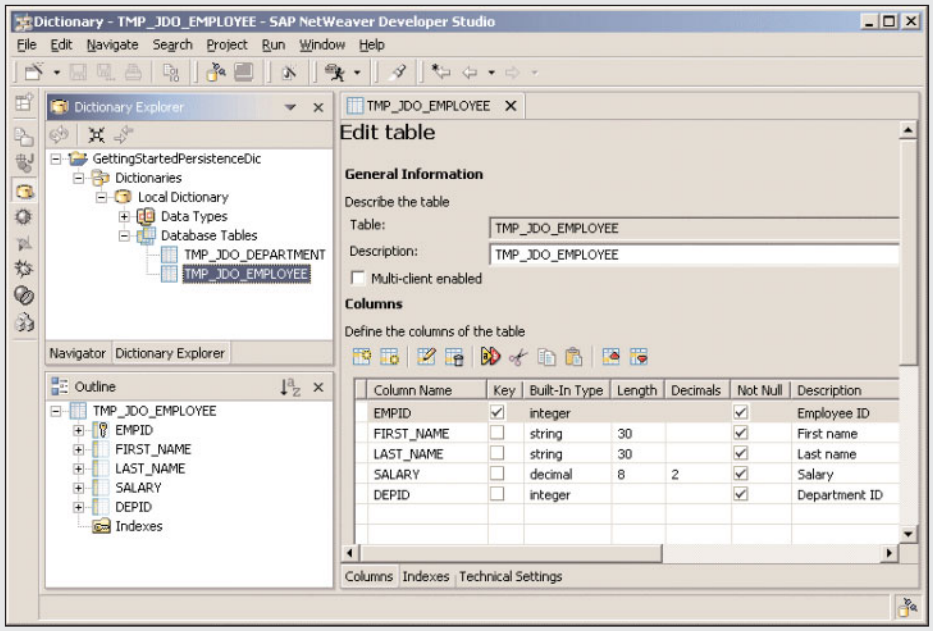


Figure 7 shows the specifications for creating the example *TMP_JDO_DEPARTMENT* table, and **Figure 8** shows the details for creating the example *TMP_JDO_EMPLOYEE* table.

Figure 9 Mapping of the Department Class

```
?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map SYSTEM "map.dtd">
<map version="1.0">
  <package name="jdoexample.persistencelogic">
    <class name="Department">
      <field name="depId">
        <column name="DEPID" table="TMP_JDO_DEPARTMENT"/>
      </field>
      <field name="name">
        <column name="NAME" table="TMP_JDO_DEPARTMENT"/>
      </field>
      <relationship-field name="employees" multiplicity="many" update="false">
         <foreign-key name="EMPLOYEE_TO_DEPARTMENT"
            foreign-key-table="TMP_JDO_EMPLOYEE"
            primary-key-table="TMP_JDO_DEPARTMENT">
            <column-pair foreign-key-column="DEPID" primary-key-
column="DEPID"/>
         </foreign-key>
       </relationship-field>
    </class>
  </package>
</map>
```

Mapping persistent JDO class attributes to database columns requires the Java type and the column's JDBC type to be compatible.[11] The JDBC type corresponds to the built-in type shown in Figures 7 and 8. The Employee class's *firstName* attribute of type *java.lang.String* is mapped to the built-in type *string*, and the *salary* attribute of type *java.math.BigDecimal* is mapped to the built-in type *decimal*, for example.

Mapping attributes of types such as *String* or *BigDecimal* is straightforward — each attribute corresponds to a single column. This is called "primitive type mapping." Typically, all primitive type attributes of a class are mapped to the same table. Mapping attributes that form a relationship to other classes is called "relationship mapping."

Mapping metadata information for a JDO class is provided in a *.map* XML file, which you add to the EJB module project's *ejbModule* folder as part of the *jdoexample.persistencelogic* package, just as you added the *.jdo* files. You must create this file manually using the XML editor in SAP NetWeaver Developer Studio. The file location and the naming conventions are the same as for the *.jdo* persistence metadata file — there is one *.map* file for each class, with the same name, and the *.map* file is located in the same folder as its associated class and persistence metadata files (in this case, *ejbModule/jdoexample/persistencelogic*).

Figure 9 shows the mapping for the example Department class. (You will find the Employee class's *.map* file in the download at **www.SAPpro.com**.)

In addition to having the same package and class element as the corresponding *.jdo* file, for each of the fields declared persistent in the *.jdo* file, there must be a corresponding field element in the *.map* file that will

[11] See the SAP NetWeaver Developer Studio help for a complete discussion of Java and JDBC type compatibility.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

define the columns to which a persistent Java class attribute is mapped.

If you map a primitive type attribute (such as *depId*), provide the name of the attribute in the field element. Inside the field element, declare a column element that specifies the column name (*DEPID*) and the table in which the column is declared (*TMP_JDO_DEPARTMENT*), according to the names used in the Dictionary project.

The 1:*n* uni-directional relationship that is declared by the *employees* attribute of the Department class is represented by a relationship field element (*employees*) that declares the multiplicity to be *many* (remember that this field contains the set of Employee attributes), thereby expressing that this attribute will refer to the side of the relationship with cardinality *n*. The relationship field contains a foreign key element (*EMPLOYEE_TO_DEPARTMENT*) specifying the tables that hold the primary key column (primary key table *TMP_JDO_DEPARTMENT*) and the foreign key column (foreign key table *TMP_JDO_EMPLOYEE*). An additional column-pair element inside the foreign key element finally declares the foreign key column (*DEPID*) and primary key column (*DEPID*). (Be aware that we refer to foreign keys in their logical sense: a column that holds the value of an identifying primary key of another table.)

> ✓ ***Tip***
>
> *A checker tool is also available as part of an SAP NetWeaver Developer Studio plug-in. This tool cross-checks the correctness of the complete model (Java object model, persistency metadata, mapping metadata, and dictionary metadata). For your reference, the download available at* ***www.SAPpro.com*** *includes some example parameters (checker.properties) for the checker tool; the details of the tool's use, however, are beyond the scope of this article. Refer to the SAP Web AS documentation in the SAP NetWeaver Developer Studio help for more information.*

You now know how to create JDO classes and perform a basic mapping between those classes and the data store. Now it is time to learn how to use JDO classes to implement business logic in a real-life application.

Integrating JDO into a Business Application

The example application we will use to demonstrate the use of JDO is a business application that follows the programming model we described earlier — that is, it is a simple application that manages two types of persistent objects: the Employee and Department JDO classes we created. The frontend of the application is a servlet named *ProcessInput*.[12] Its role is to pass user input on to the stateless session bean we created, *HRInfoBean*, which acts as a façade for the underlying Employee and Department JDO classes, and implements the logic that works with them. The result is passed back to the servlet as a value object[13] and is displayed in the browser. The entire example application, including instructions on how to implement it, is available for download at **www.SAPpro.com**.

Since the scope of our article is limited to JDOs, we'll focus primarily on how to integrate them into the example business application. In particular, we will look at:

- Identifying a persistence manager
- Implementing transaction management
- Maintaining persistent objects
- Searching for persistent objects

[12] You use the Web Module Project type in SAP NetWeaver Developer Studio to create JSP/servlet applications. While the details of the servlet implementation are beyond the scope of this article, the source code is provided in the download at **www.SAPpro.com**.

[13] This is realized by the *EmployeeVO.java* class that is provided in the download.

Figure 10 Declaring a Resource Reference in the Standard Deployment Descriptor

```
<resource-ref>
      <res-ref-name>jdo/testPMF</res-ref-name>
      <res-type>javax.resource.cci.ConnectionFactory</res-type>
      <res-auth>Container</res-auth>
      <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Identifying a Persistence Manager

So how are persistent objects accessed? This is the most important question we have to answer. To access persistent objects, to create new instances of them, and to modify and delete them, we need a "persistence manager" object. How can we get one? From a "persistence manager factory" object that instantiates the persistence manager object for you and automatically returns a reference to that object. For those of you who already have experience with JDBC or the J2EE Connector Architecture (JCA), where connection factories are used to get data store connections, this concept may seem familiar. Indeed, the persistence manager factory is the factory for the objects that enable the actual connection to the data store. The main difference in JDO is that you use the persistence manager to access any type of underlying data store (with JDBC you can only access relational databases, and with the JCA you must provide a specific resource adapter for each backend system to be accessed). The data-store-specific access is implemented by the JDO runtime and is transparent to your application.

So, the first thing we need to do is to get a persistence manager factory. When working with a standalone JDO implementation outside an application server (referred to as a "non-managed environment"), you need to construct the persistence manager factory explicitly in your code. However, the implementation of the JDO standard in SAP Web AS provides a "managed environment" for JDO-aware applications. It exposes a preconfigured persistence manager factory that you can obtain via a lookup operation in the Java Naming and Directory Interface (JNDI) registry, where it is bound to the name *jdo/defaultPMF*.

To perform the JNDI lookup, you need to declare a resource reference in the standard deployment descriptor of the J2EE application component that works with the persistence manager factory (in the example, the *ejb-jar.xml* file for the session bean *HRInfoBean*). The use of a resource reference enables you to: (1) look up the factory by a customized name instead of its predefined JNDI name; and (2) if necessary, switch to another preconfigured persistence manager factory at any time by simply modifying the deployment descriptor and redeploying the application — without changing a line of Java source code. The declaration of the resource reference defines the persistence manager factory as a resource, specifies its type, and also defines its authentication and sharing features. **Figure 10** shows the declaration for the example resource reference, which uses the customized name *testPMF*.

To map the customized name to the predefined JNDI entry, you simply use the SAP-specific deployment descriptor of the component (in the example, the *ejb-j2ee-engine.xml* file for the session bean *HRInfoBean*). **Figure 11** shows the mapping of the example customized *testPMF* name to the predefined JNDI name *defaultPMF*.

SAP NetWeaver Developer Studio provides a user-friendly way to edit the resource reference entries in the deployment descriptors. For a bean (such as our example session bean *HRInfoBean*), you use the *Enterprise Beans* tab for both the standard descriptor

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 11 Mapping a Resource Reference in the SAP-Specific Deployment Descriptor

```
<resource-ref>
      <res-ref-name>jdo/testPMF</res-ref-name>
      <res-link>jdo/defaultPMF</res-link>
</resource-ref>
```

Figure 12 Looking Up the Persistence Manager Factory

```
PersistenceManagerFactory pmf;
...
try {
      Context ctx = new InitialContext();
      pmf =
            (PersistenceManagerFactory) ctx.lookup(
                  "java:comp/env/jdo/testPMF");
      } catch (Exception e) {
            System.out.print("Failed to lookup the PMF.");
      }
...
```

(*ejb-jar.xml*) and the SAP-specific descriptor (*ejb-j2ee-engine.xml*). For a Web component, such as a servlet, you use the *Resource* tab for the standard descriptor (*web.xml*) and the *References* tab for the SAP-specific descriptor (*web-j2ee-engine.xml*).

Now we can use the resource reference name defined in Figures 10 and 11 (*testPMF*) to look up the persistence manager factory. **Figure 12** shows the lookup code included in our example session bean.

The preconfigured persistence manager factory comes with a set of standard properties, some of which you can configure programmatically. For example, the SAP JDO implementation uses the default data source of the J2EE Engine to connect to the database.[14] It defines an alias for the data source, which is the default value of the persistence manager factory's *javax.jdo.option.ConnectionFactoryName* property. You can modify this property to point to another data source object — for example, if you want to use another database.

By obtaining the persistence manager factory, you set the operational environment of your application component. Therefore, it is typically a good idea to perform this step early in the component's lifecycle. In our example, we look up the persistence manager factory in the *setSessionContext()* method of the bean (see the *HRInfoBean.java* code in the download). You can use the same model when implementing a JDO application with message-driven or entity beans — in this case, you get the reference to the persistence manager factory in the *setMessageDrivenContext()* method of the message-driven bean, or in the *setEntityContext()* method of the entity bean. In a Web component, you can use the *init()* method to obtain the *PersistenceManagerFactory* reference.

Once we have a reference to the persistence

[14] In this case, a "data source" is a Java object that is a factory for database connections (referred to as the "connection factory" in JDBC). The default data source is created when SAP Web AS is installed with the Java runtime.

manager factory, we can use it to get the persistence manager, which actually enables us to access our persistent objects.

Typically, you can obtain a persistence manager instance in a *try-catch* block. Remember that once you finish working with the persistent objects, you must close the persistence manager in the *finally* clause, even if an error occurs in the execution of the method, so that you can ensure the persistence manager is closed and that resources are released. **Figure 13** shows the *finally* clause for our example session bean.

It is possible to call the *getPersistenceManager()* method several times in a single transaction. Some of you are probably already acquainted with transactional concepts, but since transactions are a fundamental part of accessing a persistent store, we'll next pay special attention to their use in JDO.

Figure 13 Close the Persistence Manager to Release Resources

```
pm = pmf.getPersistenceManager();
// do some work
}catch (Exception e){
   e.printStackTrace();
}
finally {
   if (pm!=null && !pm.isClosed())
   pm.close();
}
```

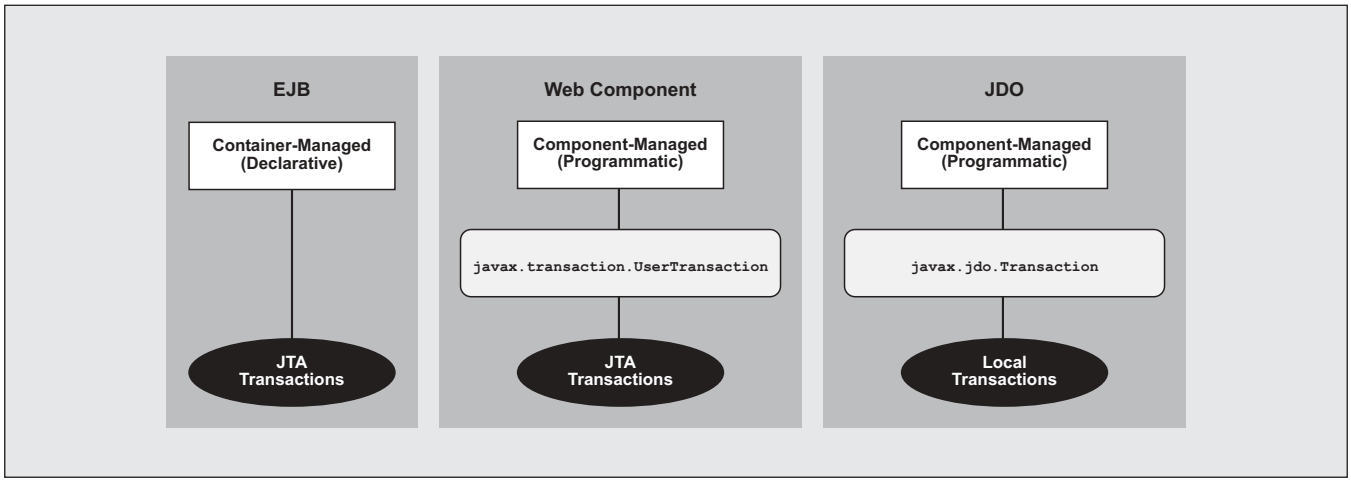
Implementing Transaction Management

When accessing a data store to modify data, it is important to keep its state consistent. You might need to perform a series of interrelated operations, which should be treated as a single unit of work. For example, you may need to withdraw money from one bank account and transfer it into another one. In addition, you might need to isolate your work so that other users cannot see what you have changed until you're done with all changes. To achieve all this, you need transactions.

The integration of the JDO implementation into SAP Web AS enables you to use both programmatic ("component-managed") and declarative ("container-managed") transactions in a JDO-aware application (see **Figure 14**).

Container-managed transactions are distributed transactions that comply with the Java Transaction API (JTA) standard. In a distributed transaction, you can access multiple resource managers simultaneously;

Figure 14 Transaction Management Options for JDO-Aware Applications



 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 15 Creating a New Department Instance in a Container-Managed Transaction

```
Public void createDepartment(int depId, String name) {
      pm = null;
      try {
            Department dep = new Department(depId);
            dep.setName(name);
            pm = pmf.getPersistenceManager();
            pm.makePersistent(dep);
      }catch (Exception e){
            e.printStackTrace();
      }
      finally {
            if (pm!=null && !pm.isClosed())
            pm.close();
      }
}
```

for example, you can change data in multiple data stores within the boundaries of a single transaction.

Since container-managed JTA transactions are only available for enterprise beans, you have another option when you need to start a distributed transaction from a Web component — you can use the *javax.transaction.UserTransaction* interface to demarcate a JTA transaction programmatically. For both container-managed and component-managed distributed transactions, the J2EE Engine in SAP Web AS does the hardest job — it synchronizes the work of the transaction branches to ensure data integrity. Additionally, the JDO API itself also provides a transaction interface — *javax.jdo.Transaction*. However, it is used for demarcating local transactions, which involve access to only a single resource manager.

Distributed Transaction Management with JTA

With container-managed transactions, you don't need to write any specific source code to demarcate the transactions. You simply declare transaction attributes for the methods of your enterprise bean. The transaction attributes are declared in the standard deployment descriptor of the bean (*ejb-jar.xml*). The EJB container then provides the transaction management functions that ensure the integrity of the persistent data. However, you still need to abide by certain rules when implementing activities that involve accessing persistent objects.

A previous article on object-oriented persistence with CMP entity beans[15] covered the basic concepts of enterprise bean development in detail, so we won't discuss how to edit your bean's deployment descriptors here. Our focus is on the use of the transaction attributes with respect to working with persistent objects. Our example illustrates a typical scenario, in which we use the JDOs behind a stateless session bean façade. In a stateless session bean, each business method is executed in a separate transaction. Therefore, we must get a persistence manager and close it at the end of each business method that works with JDO instances, as described in the previous section.

The code snippet in **Figure 15** presents a method that creates a new instance of the Department class in a container-managed transaction. (We should set the transaction attribute of the method to *Required* in the bean's deployment descriptor to ensure that the method is invoked in the context of an active transaction.)

When using component-managed transactions,

[15] "Persist Data for Your J2EE Applications with Less Effort Using Entity Beans with Container-Managed Persistence (EJB CMP)" (*SAP Professional Journal*, July/August 2004).

Figure 16 Obtaining the Current Transaction Instance

```
try {
      pm = pmf.getPersistenceManager();
      Transaction tx = pm.currentTransaction();
      tx.begin();
      Department dep = new Department(depId);
      dep.setName(name);
      pm.makePersistent(dep);
      tx.commit();
} catch (Exception e) {
      e.printStackTrace();
} finally {
      if (pm != null && !pm.isClosed()) {
            if (pm.currentTransaction().isActive())
                  pm.currentTransaction().rollback();
            pm.close();
}
```

you should generally follow the same model — simply get the persistence manager after you have started the transaction, and release it before its end.

> ✓ ***Tip***
>
> *JTA transactions are part of the J2EE standard, and we recommend using them for your J2EE application components. These transactions enable flexibility when working with backend systems in a distributed environment. Also, using JTA transactions in an SAP environment requires minimum development effort, since SAP Web AS Java provides convenient distributed transaction management mechanisms.*

Local Transaction Management with JDO

The JDO API exposes the *javax.jdo.Transaction* interface for local transaction demarcation. Each transaction instance maintains a one-to-one association with a persistence manager instance. You can obtain the transaction that is associated with the persistence manager you are working with by calling its *currentTransaction()* method, as shown in **Figure 16**. The boundaries of the transaction itself are demarcated by the invocations of the *begin()* and *commit()* or *rollback()* methods from the *Transaction* interface.

For JDO local transactions, we first get the persistence manager instance, then execute the transaction, and finally close the persistence manager. As you can see, this sequence stems from the specifics of the JDO API itself — you need the persistence manager to demarcate the boundaries of a local transaction.

The SAP JDO implementation in SAP Web AS also supports an optional feature called "optimistic transactions" that defines the default behavior for transactions. In optimistic transactions, the system does not lock the data while it is being modified. Any changes made by other transactions during that same time period will be detected at commit time and transaction rollback will be triggered to prevent data inconsistency. The isolation between the transactions is defined by the isolation level of the database connection that the persistence manager factory uses. For the connections obtained from the default data source, the isolation level is *TRANSACTION_READ_UNCOMMITTED*.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Since the JDO runtime implements the optimistic transactions feature by default, a logical question is how to disable optimistic transactions — for example, to make sure that no other applications can modify the same data at the same time. In SAP Web AS you simply set the *javax.jdo.option.Optimistic* property of the persistence manager factory to *false* and use explicit locking, which is explained next.

Explicit Locking

If you need to ensure that the data your application reads or modifies is not being changed in another transaction at the same time, you must use explicit locking. This implies that you must explicitly define which data you want to lock when executing read or write operations.

The JDO runtime locks the relevant relational structures in the database, not the JDO instances themselves. You can lock data in three modes:

- *MODE_SHARED*
- *MODE_EXCLUSIVE_CUMULATIVE*
- *MODE_EXCLUSIVE_NONCUMULATIVE*

The first mode enables a shared lock among several threads, which only read the data. It guarantees that data is not changed at the time the threads read it. The other modes are appropriate for write operations, since they guarantee that only a single thread accesses the locked data. "Cumulative" implies that the same transaction may use the lock several times. The system releases the locks automatically for you at the end of either the transaction or the user session, which depends on the lifetime parameter that you have set (*LIFETIME_TRANSACTION* or *LIFETIME_USERSESSION*).

> ✓ ***Tip***
>
> *Explicit locking is time-consuming and hence increases response times, but it guarantees a higher level of isolation and solves possible problems with read and write operations in concurrent environments.*

In **Figure 17**, we have implemented local transaction demarcation and explicit locking in the *changeDepartmentName()* method of our example session bean. While modifying the data, we impose an exclusive cumulative lock, which is released at transaction commit.

Figure 17 Implementing Local Transaction Demarcation and Explicit Locking

```
import com.sap.jdo.Locking;
import com.sap.jdo.SAPJDOHelper;
...
public void changeDepartmentName(int depId, String newName) {
            pm = null;
            try {
                  pm = pmf.getPersistenceManager();
                  Transaction tx = pm.currentTransaction();
                  tx.setOptimistic(false);
                  Locking locking = SAPJDOHelper.getLocking();
                  tx.begin();
                  Department department =
                        (Department) pm.getObjectById(new
                        Department.Id(depId),
```

(continued on next page)

Figure 17 (continued)

```
false);
                    try {
                          locking.lock(
                                Locking.LIFETIME_TRANSACTION,
                                pm,
                                department,
                                Locking.MODE_EXCLUSIVE_CUMULATIVE);
                    } catch (Exception e) {
                          e.printStackTrace();
                    }
                    department.setName(newName);
                    tx.commit();

              } finally {
                    if (pm != null && !pm.isClosed()) {
                          if (pm.currentTransaction().isActive())
                                pm.currentTransaction().rollback();
                          pm.close();
              }
        }
}
```

Maintaining Persistent Objects

Now that you know how to obtain a persistence manager instance and handle transactions, we can move on to what you can actually do with persistent objects. Your persistence manager is your link to the data store; its role is similar to the role of the database connection in JDBC. The persistence manager also handles the persistent objects for you, as each instance of a persistent object is associated with a persistence manager instance. In the next sections, we will show you some examples of the ways in which you can use your persistence manager to create, retrieve, update, and delete persistent objects in the data store.

Creating Persistent Objects

First, let's look at how to create a new persistent object by constructing an instance of the relevant persistence-capable class. The newly created object is transient; that is, the object does not yet exist in the data store, and doesn't have an established association with a persistence manager. To make the object persistent, simply call the persistence manager's *makePersistent()* method and the object is stored in the database at transaction commit:

```
Department dep = new
  Department(depId);
dep.setName(name);
pm = pmf.getPersistenceManager();
pm.makePersistent(dep);
pm.close();
```

> ✓ ***Tip***
>
> *The persistence manager's makePersistent() method must be called in an active transaction only.*

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Retrieving Persistent Objects

The concept of object identity in JDO enables you to easily retrieve a uniquely identified instance of a persistence-capable class:

```
pm = pmf.getPersistenceManager();
Department department =
  (Department) pm.getObjectById(new
Department.Id(depId), false);
emp.setDepartment(department);
department.addEmployee(emp);
```

Here you can see how the object identity concept, which we discussed earlier in the article when we created the JDO classes, is used in practice. A new object identity is constructed that corresponds to the primary key of the persistence-capable class — in this case, the *depID* primary key of the Department class. However, it is also possible to retrieve the identity of a JDO instance by calling the *getObjectId()* method of the persistence manager. You might need to use this approach if you want to get exactly the same instance later on (to delete a particular instance of a class, for example). Then you can use the returned result as a parameter to the *getObjectById()* method.

The second parameter of the *getObjectById()* method defines how the JDO runtime validates the instance. If you set the validation flag to *true*, the system checks for the existence of the relevant persistent object in the data store. If the flag is set to *false*, you get a "hollow" instance; that is, the instance is not populated with values taken from the data store, and the JDO runtime does not check at this point to see if the persistent object really exists. As you can imagine, you will get an exception if you later try to access the fields of a non-existing object.

What if you need to get *all* instances of a persistence-capable class? JDO enables you to do this by using an "extent," which is a very useful feature, as you will see later when we discuss queries. You shouldn't perceive a *javax.jdo.Extent* instance as a collection of persistent objects — it is rather an object that encapsulates information about the persistence-capable class and its subclasses (if any), and thereby provides you with a handle to all instances of these classes stored in the database.

To construct an extent, use the *getExtent()* method of the persistence manager, with the persistence-capable class passed as a parameter:

```
Extent employeeExtent =
  pm.getExtent(Employee, true);
```

Optionally, you can choose to include the instances of the existing subclasses in the extent, which you declare using the boolean flag *subclasses*.

Constructing an extent does not actually involve any data store operations. The data store is accessed when you start iterating the extent as follows:

```
Iterator iter =
  employeeExtent.iterator();
while (iter.hasNext()) {
      Employee emp = (Employee)
        iter.next();
      System.out.println(
        emp.getFirstName() + " " +
        emp.getLastName());
}
employeeExtent.close(iter);
```

Then the instances of the persistence-capable class are actually retrieved. An iterator is provided by the *iterator()* method of the extent, and should be closed when iteration is complete.

Updating a Persistent Object

Using the *getObjectById()* method, you can retrieve a particular persistent object that contains fields you want to update:

```
pm = pmf.getPersistenceManager();
Department department =
  (Department) pm.getObjectById(new
Department.Id(depId), false);
department.setName(newName);
pm.close();
```

Here, the persistence-capable classes expose setter methods for each persistent field, so to change the values of a field, simply call the relevant setter method with the new value as a parameter. The JDO runtime transparently applies the changes to the object in the database at transaction commit.

> ✓ ***Tip***
>
> *The default persistence manager factory does not allow non-transactional write operations — all changes to persistent objects must be done in the context of an active transaction.*

Deleting a Persistent Object

It is as easy to delete a persistent object as it is to create it. Again, you only need to tell the persistence manager what you want to do — that's it! Of course, you need to specify which object should be deleted, and you do that using either a query or a *getObjectById()* method, as follows:

```
Employee emp =
  (Employee)pm.getObjectById(new
  Employee.Id(empId), false);
Department dep =
  emp.getDepartment();
dep.removeEmployee(emp);
pm.deletePersistent(emp);
pm.close();
```

As you can see, you pass the result as a parameter to the *deletePersistent()* method of the persistence manager. Similar to the update operation, the deletion of a persistent object is only possible within the boundaries of an active transaction. The object is actually deleted in the data store when the transaction commits.

Searching for Persistent Objects

As you saw in the discussions in the previous section, we need queries to retrieve, update, and delete persistent objects. JDO comes with a Java-like query language called JDO Query Language (JDOQL) that supports both static and dynamic queries. JDOQL enables you to select instances of persistence-capable classes using specified search criteria. One of the main features of JDOQL is its "data-store neutrality" — that is, you use the same API against any type of data store. At the data-store level, JDOQL queries are "translated" into the query language supported by the particular data store in use.

JDO queries are instances of the *javax.jdo.Query* interface, which exposes methods that enable you to construct and execute queries. Each query contains three required elements:

- The name of the candidate class
- A set of candidate instances to be included in the search
- Filtering criteria

For the first element, you define instances of which class and its subclasses should be included in the result of the query. For the second required element, namely the set of candidate instances, you can use either a collection or an extent. If you specify an extent, you don't need to set the class of candidates explicitly, because the extent already contains this information. If you skip this element, the query is executed against the extent of the candidate class, which you specify. The query filter is a boolean expression that is evaluated for each instance from the candidate set. The result of the query contains only those instances for which the filter evaluates to *true*. If you don't enter filtering criteria, the element defaults to *true* for all candidate instances of the specified class.

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.

In addition to the required elements, you may also provide parameters or variables to the query, as well as set result ordering.

In the next sections, we will show you how to construct a query, execute it, and process the results.

Step 1: Construct the Query

The first thing you should know about JDO queries is how to construct one. This is done by calling the *newQuery()* method of the persistence manager:

```
Query queryExample = pm.newQuery();
```

The persistence manager interface exposes several methods for instantiating a query. Here, we use a method without parameters to construct an empty query so we can set its elements. The persistence manager also enables you to initialize a query that passes its elements directly as parameters to the *newQuery()* method, or using the elements of another query. In the following code snippet, we use an extent and a filter to construct a new query that returns all employees with the first name "John":

```
String filter = "firstName ==
  \"John\" ";
Query queryEmployees =
  pm.newQuery(employeeExtent,
  filter);
```

Here, we use a simple expression as a filtering criterion, but you can also create more complicated expressions using equality, comparison, boolean, and arithmetic operators. You can set the values in the filter dynamically using parameters as well. The *Query* interface provides the *declareParameters()* method, which enables you to specify different values for the filtering criteria. You can see this in the following coding, where we declare the department ID as a parameter of the query to retrieve an object that corresponds to a dynamically specified ID:

```
public Employee[]
  getEmployeesFromDep(int
  departmentId) {
...
String filter = "department.depId
  == id";
Query query =
  pm.newQuery(Employee.class,
  filter);
query.declareParameters("int id");
Collection col = (Collection)
  query.execute(new
  Integer(departmentId));
...
```

In a query, you can also use variables to select from a collection of instances that are related based on certain criteria. You declare the variable by calling the *declareVariables()* method of the *Query* interface, and use it as a parameter for the *contains()* method of the collection, as follows:

```
String filter =
  "employees.contains(emp) &&
  emp.salary > 1000";
Query query =
  pm.newQuery(departmentExtent,
  filter);
query.declareVariables("Employee
  emp");
Collection result =
  (Collection)query.execute();
```

JDOQL also enables you to order the result of the query with the *setOrdering()* method, which functions similarly to the *ORDER BY* clause in SQL.

Step 2: Execute the Query and Process the Result

Once you have constructed the query and initialized all of its elements, you can execute it by calling the *execute()* method of the *Query* interface. The result is always a collection, and must be cast accordingly:

```
String filter = "department.depId
  == id";
Query query =
  pm.newQuery(Employee.class,
  filter);
query.declareParameters("int id");
Collection result =
  (Collection)query.execute();
Iterator iter = col.iterator();
      while (iter.hasNext()){
            Employee emp =
              iter.next();
            System.out.println(
              emp.getLastName());
      }
query.close(col);
```

To process the result, you need to iterate over the collection. At the end of the operation, you release the used resources by explicitly closing the result.

Ready for JDO?

So when is JDO more appropriate than other Java persistence technologies provided by SAP Web AS? To answer this question, we need to take a look at JDO's main advantages, which are:

- JDO employs an object-oriented paradigm, combined with an easy-to-use API for domain-model-centric applications.
- JDO enables data store neutrality and application portability.
- The SAP JDO implementation provides a powerful object-relational mapping mechanism.

However, in certain cases you might need a relational-oriented approach. For example, this might be true in the case of data-centric, processing-intensive, performance-sensitive (batch) applications, due to the bookkeeping overhead caused by object-relational persistence frameworks in general. Another possible scenario is when using extremely complex relational legacy schemas, which cannot be handled by the object-relational mapping capabilities of the persistence framework. You might also need to use specific SQL queries that cannot be expressed in a framework's query language like JDOQL. In such cases you may want to consider using other technologies, such as OpenSQL/SQLJ.[16] If you are not constrained by such scenarios, however, JDO is the easiest and most efficient choice for Java persistence.

Markus Küfer holds a degree in medical informatics from the University of Heidelberg, Germany. He joined SAP in 2000 and worked on Java technologies for technology and application frameworks. With the advent of JDO 1.0, Markus joined the initial core team that designed the architecture of the SAP JDO implementation and was responsible for the object-relational mapping and integration with the SAP J2EE Engine. Currently, he is concerned with SAP J2EE Engine kernel development, focusing on the distributed configuration repository infrastructure, and serves as SAP's representative in the JDO Java Specification Request expert group. Markus can be reached at markus.kuefer@sap.com.

Zornitsa Yankulova holds a master's degree in international relations from the University of National and World Economy in Sofia, Bulgaria. In 2000, she joined InQMy (which later became SAP Labs Bulgaria) as an information developer. Zornitsa's main focus for the past two years has been the persistence and backend connectivity framework of SAP Web AS Java, including the JDO implementation in the SAP J2EE Engine. She can be reached at zornitsa.yankulova@sap.com.

[16] See the article "Achieving Platform-Independent Database Access with Open SQL/SQLJ — Embedded SQL for Java in the SAP Web Application Server" (*SAP Professional Journal*, January/February 2004).

 ©2004 SAP Professional Journal. Reproduction prohibited. All rights reserved.