USING ORACLE OBJECTS IN SQLJ PROGRAMS

Ekkehard Rohwedder, Oracle Corporation

INTRODUCTION

The Oracle8*i* data server supports the definition of object types and the storage of object instances in the database. Objects can be manipulated using Java programmatic interfaces such as JDBC and SQLJ. This paper examines and illustrates object manipulation through static SQL statements in SQLJ programs. We describe a general mechanism with which any kind of SQL data may be read and written by Java in fully user-customizable fashion. As an example, we show how users can provide their own customized mapping from RAW columns in SQL to serialized Java objects. The same mechanism is then employed to create mappings from SQL object types to Java classes. We show how the JPublisher tool (formerly Object Type Translator) assists in the generation of customized Java class definitions for these types. Several examples illustrate these concepts from the creation of the object types in the database, to the examination of the generated Java wrappers, and to using these classes in SQLJ programs for querying and updating objects in the database. We also compare the SQLJ and JDBC approaches. The underlying representation of objects is shared between both APIs, which makes all Java class declarations for object types fully interchangeable. On the other hand, SQLJ users benefit from concise code -object types do not need any special syntactic treatment- as well as from translate-time checking of SQL syntax, semantics, and type compatibility.

After giving a brief introduction to SQLJ, we discuss in section 2 the basic representation of SQL data. A general customization mechanism for mapping SQL data to Java is described in section 3, and we provide an in-depth example mapping RAW data to Java serializable objects. In section 4 we focus on the representation of SQL object types, REFs, and collection types in Java. The JPublisher tool is also discussed, as well as the Java code generated by it. Several examples show SQL code for object creation, and the SQLJ code for corresponding object manipulation. The final section concludes with an assessment of our approach and a comparison between using the JDBC and SQLJ APIs for exploiting SQL objects.

1. SQLJ-OVERVIEW

This section summarizes the SQLJ standardization effort, as well as the basic features of the SQLJ translator program, see also [1]. We examine the benefits of using SQLJ, and survey the basic mechanism that permits the SQLJ translator to support Oracle8-specific types.

1.1 STANDARDIZATION

SQLJ is a standardization effort that defines the interoperability between Java and SQL. It is driven by several vendors, including Oracle, Compaq/Tandem, IBM, Sybase, Microsoft, Informix, and Sun Microsystems. *SQLJ Part 0* specifies the embedding of static SQL statements in the Java language and has been submitted to ANSI as draft X3H2 98-227. We refer to this document when we talk about the standard for *Embedded SQL in Java*.

1.2 SQLJ TRANSLATOR

A reference implementation of the *SQLJ translator,* written in pure Java, has been created by Oracle and its partners and is publicly available. The translator converts SQLJ programs with embedded SQL statements into Java programs with calls to the *SQLJ runtime*. This is similar to the Pro*C precompiler translating SQL statements embedded in C. The SQLJ translator employs an open architecture, permitting it to be integrated into different development environments, such as graphical IDEs as well as the Oracle8*i* server side JavaVM. Additionally, SQLJ can support *SQL checking* and *SQL runtime customizations* for arbitrary databases through Java plug-ins.

Developing an SQLJ application is a straightforward process that consists of four steps after the code is written:

1. Translation of SQLJ source files with the SQLJ translator. This generates new Java source files with calls to the SQLJ runtime, as well as additional *SQLJ profile files* containing all the information about the static SQL statements that were found in the SQLJ source.
2. Compilation of Java sources with a Java compiler.

3. Customization of the generated SQL profiles for improved runtime performance and vendor-specific features.
4. Running the application, using the SQLJ runtime library.

Usually, steps 1 to 3 are automatically performed by the SQLJ translator by transparently invoking a Java compiler in a subprocess. At translation time, the static SQL statements in the program can be checked against a given database schema. The SQLJ runtime can either use an arbitrary JDBC driver, or it can be implemented separately without any relationship to JDBC. The Oracle SQLJ runtime uses the Oracle JDBC driver.

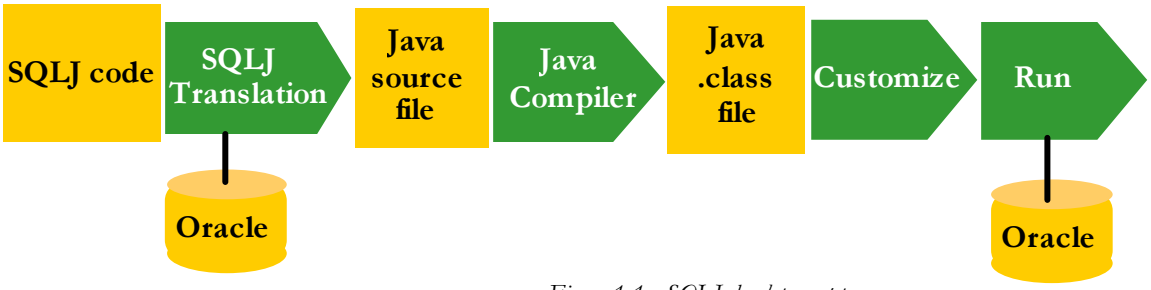

Figure 1-1 - SQLJ development process

1.3 Advantages of SQLJ

Assume you wanted to create Java applications that require database access. Should you consider SQLJ rather than the JDBC API for database connectivity? The following are *benefits* provided by SQLJ.

1. a much more concise syntax for database access,
2. strong typing, e.g. of queries through *iterator types*, and of connections through *connection context types*,
3. checking of syntax and semantics of all SQL statements,
4. full binary portability of SQLJ programs through multi-vendor ANSI standardization, including the SQLJ runtime; additional uniformity is provided through the translator reference implementation,
5. plug-in architecture facilitates tool support for SQLJ. For example, both, Oracle JDeveloper and the Oracle8*i* JavaVM integrate the SQLJ translator,
6. *customization of the runtime profile*, which describes the static SQL statements, encompasses:
    - performance enhancements, e.g. through caching or pre-compilation,
    - added functionality, such as the support of vendor-specific types,
    - migration support, and
    - debugging or logging functions.
7. multi-vendor standard for embedded static SQL in Java.

Even for applications that contain some dynamic SQL mixed in with static SQL statements using SQLJ may be advantageous, since it is possible to mix-and-match SQLJ and JDBC in the same application. Specifically, connections as well as result sets can be shared between both APIs. SQLJ and JDBC constitute *complementary* technologies and strengths. While SQLJ provides more concise syntax for static SQL, it relies on the established JDBC interface for embedding dynamic SQL in Java.

It should also be noted, that the SQLJ runtime is immediately available on top of any JDBC driver. This means that SQLJ applications can be deployed wherever JDBC is deployed, such as on a thin client, a fat client, or in the server.

1.4 SQLJ Support for Oracle Types

SQLJ can support Oracle8 types through an Oracle-specific customization of the SQLJ runtime profile. This customization will be automatically performed when you use the SQLJ translator that is provided with the Oracle8*i* database. In this process, runtime calls to standard JDBC entry points, such as `getObject()` and `setObject()` are replaced with calls to Oracle's JDBC API.

We do expect the SQLJ specification to evolve in the future to encompass structured SQL3 types, such as those that were introduced in JDBC 2.0, and are supported preliminarily in the Oracle8*i* JDBC drivers.[1] Currently, support for Object Types can only be provided as a vendor-specific extension.

The Object Type support that is outlined here is based on the efficient, direct representation of the SQL data. The next section provides a description of this representation.

2. SQL Data Types and Java Objects

Oracle's 8.1 JDBC driver [3] introduces a new package `oracle.sql` that contains Java classes corresponding to all existing SQL types. In addition to providing efficient low-level representation and manipulation of SQL data, all of these types also permit full customization for user-provided Java classes. This customization mechanism is described in section 3, and it also forms the basis for reading and writing SQL Object Types as instances of Java classes.

2.1 SQL Types and oracle.sql Classes

The table below outlines the correspondence between SQL types and `oracle.sql` classes. All Java representations of SQL data in the second column subclass the Java type `oracle.sql.Datum`. All of these classes also hold a binary representation of the SQL data in the form of a `byte` array. It is returned by the `getBytes()` method.

SQL Type	*Java representation*	*JDBC Typecode*[2]	*Associated with SQL Type Name*
All Numeric Types	`oracle.sql.NUMBER`	`OracleTypes.NUMBER`	no
`CHAR, VARCHAR, VARCHAR2`	`oracle.sql.CHAR`	`OracleTypes.CHAR`	no
`DATE`	`oracle.sql.DATE`	`OracleTypes.DATE`	no
`RAW`	`oracle.sql.RAW`	`OracleTypes.RAW`	no
`ROWID`	`oracle.sql.ROWID`	`OracleTypes.ROWID`	no
`CLOB`	`oracle.sql.CLOB`	`OracleTypes.CLOB`	no
`BLOB`	`oracle.sql.BLOB`	`OracleTypes.BLOB`	no
`BFILE`	`oracle.sql.BFILE`	`OracleTypes.BFILE`	no
Object Type	`oracle.sql.STRUCT`	`OracleTypes.STRUCT`	yes
`REF` Type	`oracle.sql.REF`	`OracleTypes.REF`	references an Object Type
Varray or Nested Table	`oracle.sql.ARRAY`	`OracleTypes.ARRAY`	yes

Table 2-1 - Representation of SQL Data in Java

The CLOB, BLOB, and BFILE classes encapsulate locator types, and provide a stream API for accessing and manipulating the actual data. The STRUCT, ARRAY, and REF classes correspond to named types, varrays/nested tables, or references to named types, respectively - we will explain these SQL types further in section 4. The classes NUMBER, CHAR, DATE, RAW, and ROWID provide several conversions from the SQL data to native Java types.

The Oracle JDBC driver provides additional methods, such as `getNUMBER()` and `setNUMBER()`, etc. for reading and writing these `oracle.sql` types on Oracle result sets and statements.

2.2 SQLJ Support for oracle.sql Classes

SQLJ also supports all of the types in `oracle.sql` directly. Users may declare host variables of these types and use them to materialize result set columns, to pass values in DML statements, or in stored functions, procedures, and PL/SQL blocks.[3]

---

[1]Oracle SQLJ currently does not support the JDBC 2.0 types. With full Oracle JDBC 2.0/JDK 1.2 driver availability, we expect that SQLJ support will be extended to all JDBC 2.0 SQL types, with the interface `java.sql.SQLData` playing a similar role as `oracle.sql.CustomDatum` described in section 3.

```
oracle.sql.NUMBER number;
oracle.sql.NUMBER no = new oracle.sql.NUMBER(7902);
#sql { SELECT COMM INTO :number FROM EMP WHERE EMPNO = :no };
if (number == null) {
   System.out.println("Commission is NULL");
} else {
   System.out.println("Commission is:" + number.doubleValue());
}
```

Figure 2-1 - Using oracle.sql.NUMBER in SQLJ

3. CUSTOMIZING SQL DATA TYPES

Oracle's SQLJ translator (as well as the Oracle 8.1 JDBC driver) support a mechanism that permits users to fully customize the way in which an `oracle.sql` datum is read from or written to the database - for more details, see also [2]. In effect, users can provide their own customized wrappers for reading and writing SQL data. These wrappers implement the `CustomDatum` interfaces described below.

3.1 CUSTOMDATUM INTERFACES

All data that is passed to or from the database is in form of an `oracle.sql.Datum`. A user will be providing her own customized Java data through implementing the `CustomDatum` interface. In order to send a `CustomDatum` to the database, it must be convertible to an `oracle.sql.Datum` via a public `toDatum()` method.

```
interface oracle.sql.CustomDatum
{
   oracle.sql.Datum toDatum();
}
```

Figure 3-1 - oracle.sql.CustomDatum interface

Additionally, given an appropriate `oracle.sql.Datum`, we need to be able to construct an instance of the user's `CustomDatum`. This property is captured in the `create` method of the `oracle.sql.CustomDatumFactory` interface.[4]

```
interface oracle.sql.CustomDatumFactory
{
   oracle.sql.CustomDatum create(oracle.sql.Datum d);
}
```

Figure 3-2 - oracle.sql.CustomDatumFactory interface

We still have to connect both of these interfaces. We do so by requiring that the user's implementation of the `CustomDatum` interface also provide a corresponding `CustomDatumFactory` which may be obtained by the static method:[5]

```
public static oracle.sql.CustomDatumFactory getFactory();
```

Most likely, wrappers implementing this interface will be used in conjunction with an `oracle.sql.STRUCT` (in the case of Object Types), an `oracle.sql.REF` (an SQL reference to an Object Type), or an `oracle.sql.ARRAY` (for SQL varrays and nested tables) - this is detailed in section 4.

However, it can occasionally be useful to provide customized wrappers for one or more of the other types as well. Such wrappers might be used, for example

- to perform encryption and decryption of data,

---

[2] `OracleTypes` refers to `oracle.jdbc.driver.OracleTypes`. The first four typecodes in the table are identical to corresponding values in `java.sql.Types`. They are, respectively, `Types.NUMERIC`, `Types.CHAR`, `Types.DATE`, and `Types.BINARY`.

[3] There is one caveat: types associated with SQL names, such as STRUCT, REF, and ARRAY, represent *weak types*, and may not be used as OUT or INOUT host variables in stored procedures, functions, or PL/SQL blocks.

[4] For pedagogical purposes we omit two minor details in these interfaces. Read on to discover them.

[5] Since interfaces cannot declare `static` methods, we could not list this requirement in the `CustomDatum` interface.

- to perform validation of data,
- to perform logging of values that have been read or are being written,
- to parse character columns (e.g. character fields containing URL information) into smaller components, or to map character strings into numeric constants,
- to perform mapping of data (e.g. a DATE field) into more desirable Java formats (e.g. `java.util.Date`),
- to serialize and deserialize Java objects into and out of RAW fields, etc.

We will expand on the last bullet and show how users can define a customization for RAW columns that provides automatic serialization and deserialization of Java objects.

3.2 EXAMPLE: SERIALIZATION OF JAVA OBJECTS

The user's class is called `SerializableDatum` and defined in the file `SerializableDatum.java`. The program uses classes from `java.io`, `java.sql`, `oracle.sql`, and `oracle.jdbc.driver`, however we do not explicitly show the import statements here. The skeleton of this program follows the `CustomDatum` interface outlined above.

```
public class SerializableDatum implements CustomDatum
{
  Client_methods_for_constructing_and_accessing_the_Java_Object

  public Datum toDatum(OracleConnection c) throws SQLException
  {
    Implementation_of_toDatum
  }

  public static CustomDatumFactory getFactory()
  {
    return FACTORY;
  }
  private static final CustomDatumFactory FACTORY =
    Implementation_of_a_CustomDatumFactory_for_SerializableDatum ;

  Constructing_SerializableDatum_from_oracle.sql.RAW

  public static final int _SQL_TYPECODE = OracleTypes.RAW;
}
```

Figure 3-3 - Skeleton of SerializableDatum class

Here the `getFactory` method simply returns a static member that implements the `CustomDatumFactory` interface. We also notice that the `toDatum` method on the `CustomDatum` interface actually takes an Oracle JDBC connection as an argument (this is necessary to ensure proper type checking and conversion for named types at runtime) - we glossed over this detail in *Figure 3-1*.

The declaration above contains an additional field `_SQL_TYPECODE`, designating the actual `oracle.sql` type that we expect to read and write. The SQLJ translator expects the typecode to be present and employs it to determine compatibility between the user-specified Java type and the SQL type in the database. These codes (as well as additional information required in certain cases) are also used by the JDBC runtime.

Next we provide the definitions of the client methods for creating a `SerializableDatum`, populating it with a Java object, and retrieving a Java object.

Client_methods_for_constructing_and_accessing_the_Java_Object ::=

```
private Object m_data;
public SerializableDatum()      { m_data = null; }
public void setData(Object data){ m_data = data; }
public Object getData()         { return m_data; }
```

Figure 3-4 - Client methods of SerializableDatum

The implementation of `toDatum` must return a serialized representation of the object in the `m_data` field as an `oracle.sql.RAW` instance. You will see the well-known steps required for serialization of a Java object. A `RAW` can immediately be constructed from a `byte` array.

Implementation_of_`toDatum` ::=

```
try {
  ByteArrayOutputStream os = new ByteArrayOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream(os);
  oos.writeObject(m_data);  oos.close();
  return new RAW(os.toByteArray());
} catch (Exception e){
  throw new SQLException("SerializableDatum.toDatum: "+e.toString());
}
```

Figure 3-5 - Serializing a Java object to an oracle.sql.RAW

In the opposite direction, we must program the conversion of an `oracle.sql.RAW` instance to a Java object. Now we have to perform deserialization steps. Rather than just return an `Object`, we construct a new instance of `SerializableDatum` with the `data` field instantiated.

Constructing_`SerializableDatum`_from_`oracle.sql.RAW` ::=

```
private SerializableDatum(RAW raw) throws SQLException {
  try {
      InputStream rawStream = new ByteArrayInputStream(raw.getBytes());
      ObjectInputStream is = new ObjectInputStream(rawStream);
      data = is.readObject();
      is.close();
  } catch (Exception e) {
    throw new SQLException("SerializableDatum.create: "+e.toString());
} }
```

Figure 3-6 - Constructing an instance of SerializableDatum from oracle.sql.RAW

Finally, we add the last puzzle piece to the program by providing an implementation instance of the `CustomDatumFactory` interface. We create the implementation in form of an anonymous class.

Implementation_of_a_`CustomDatumFactory`_for_`SerializableDatum` ::=

```
new CustomDatumFactory() {
    public CustomDatum create(Datum d, int sqlCode)
    throws SQLException
    {
      if (sqlCode != _SQL_TYPECODE)
      {
        throw new SQLException("SerializableDatum: invalid SQL type "+sqlCode);
      }
      return (d == null) ? null : new SerializableDatum((RAW)d);
 }  }
```

Figure 3-7 - Implementation of CustomDatumFactory

Note that we previously omitted the fact that `CustomDatumFactory.create` takes a second argument of type `int` with the JDBC typecode of the `oracle.sql.Datum`. We also see in this example how this code is used for type checking.

We conclude the example by demonstrating how the user-defined type can be immediately used in an SQLJ program. The display below shows the creation of a table with a `RAW` column. The corresponding SQLJ code fragment inserts a Java object

(in the particular example, an array of `Object`) into this table.

```
CREATE TABLE PERSDATA (NAME VARCHAR2(20) NOT NULL, INFO RAW(2000));

SerializableDatum pinfo = new SerializableDatum();
pinfo.setData(new Object[] { "Some objects", new Integer(51), new Double(1234.27) });
String pname = "MILLER";
#sql { INSERT INTO PERSDATA VALUES( :pname, :pinfo) };
```

Figure 3-8 - Inserting a SerializableDatum into a table

We can also create an iterator for traversing PERSDATA that returns a `SerializableDatum` column. Iterator definition and traversal consists of the following steps.

1. Declaration of an iterator type. Here we declare `PersIter`, a named iterator type with the columns `name` and `info`.
2. Definition of an iterator instance. We call it `pcur` in our example.
3. Assigning the result of a query to the iterator instance. Since we used a *named* iterator, the binding of columns will be performed by name, and the column order in the query does not matter.
4. Traversing the rows of the query result via the `next()` method. The values of the columns `name` and `info` are available through corresponding accessor methods on the iterator `pcur`.

```
#sql iterator PersIter (SerializableDatum info, String name);
…
PersIter pcur;
#sql pcur = { SELECT * FROM PERSDATA WHERE info IS NOT NULL };
while (pcur.next())
{
   System.out.println("Name:" + pcur.name() + "Info:"+ pcur.info());
}
```

Figure 3-9 - Using SerializableDatum in an iterator

From the point of view of SQLJ, a `SerializableDatum` may be used in SQLJ code, whenever the type RAW is expected in SQL. This example also exhibits the following limitations in reading and writing Java objects. We may exceed the size of the `RAW` column into which we write. Moreover, Java must also serialize all objects referenced by the object contained in `SerializableDatum.m_data`. Thus, if Java objects are shared, many copies of them may actually be deserialized. Later, when retrieving the serialized data back to Java, sharing will have been broken. In these cases it is better to model objects directly in SQL, such as with the Oracle8 Object Types and REFs that are described in the next section.

4. Publishing SQL Object Types and Collections to Java

We briefly survey Object Types, REFs, and Collection Types in Oracle8 and examine how users may create Java wrappers for these types. Oracle provides the JPublisher tool (similar in functionality to the Object Type Translator for C) for automating much of the effort in creating the corresponding Java declarations for these types. We do give examples of using JPublisher, as well as of employing the generated Java types in SQLJ programs.

4.1 Object Types

An object type is similar to a SQL3 named row type and consists of one or more attributes that define the structure of an object.[6] Objects are useful when representing real-world entities which may have a complex set of attributes. Once an object is created, it can be stored in or accessed from relational tables as easily as any of the basic SQL data types such as a NUMBER or CHAR. The SQL declaration in the display below defines the Object Type PERSON. Subsequently, the constructor PERSON is invoked on the attribute values of the type to create a new object instance of PERSON type.

```
CREATE TYPE PERSON AS OBJECT
( FIRSTNAME  VARCHAR2(15),
```

[6]Oracle8 Object Types actually constitute an extension of SQL3 Named Row Types. They may have member methods or functions that model the behavior of an object. Since the Java publishing mechanism currently ignores these methods, we will also not deal with them in this paper.

```
  LASTNAME   VARCHAR2(30),
  BIRTHDAY   DATE
);
CREATE TABLE EMPS (PERS PERSON, EMPNO INTEGER, SAL NUMBER(7,2));

DECLARE P PERSON;
BEGIN
  P := PERSON('Albert', `Einstein', TO_DATE('14-MARCH-1879'));
  INSERT INTO EMPS VALUES(P, 1001, 5000.0);
END;
```

Figure 4-1 Declaring and instantiating the type PERSON

4.2 PUBLISHING JAVA WRAPPERS FOR OBJECT TYPES

SQL Object Type values are materialized in Java as instances of the class `oracle.sql.STRUCT`. A `STRUCT` contains a field `values` which is an array with elements of type `oracle.sql.Datum` holding the actual values of all of the attributes of the Object Type.

Given this information, we could now go ahead and manually write a `CustomDatum` wrapper for a given SQL Object Type. However, this endeavor is better left to a tool - JPublisher. The JPublisher is given SQL Object Types and creates the source code for a corresponding Java wrapper classes. The attributes of the SQL type can be materialized in one of two styles:

1. the default JDBC mapping from SQL types to Java types, or
2. the `oracle.sql` representation.

We have to communicate to JPublisher the names of wrapper classes that are to be generated, as well as the SQL Object Types from which they originate. This is accomplished via a *typefile* that is passed to JPublisher. The syntax of the entries in the typefile is: `TYPE` <*SQLType*> `AS` <*JavaType*>. In the example we request that the wrapper class be called `MyPerson`.

Contents of the file `types.in`:

```
TYPE PERSON AS MyPerson
```

Command line invocation of JPublisher:

```
jpub -user=scott/tiger -typefile=types.in
```

Figure 4-2 - Invocation of JPublisher for the ObjectType PERSON

JPublisher will now generate two files, `MyPerson.java` with the Java wrapper for PERSON objects, and `MyPersonRef.java`, with the Java wrapper for (strongly typed) REFs to PERSON. Below we show the content of `MyPerson.java` (after removing implementation details).The second file will be examined in the next section, where we discuss REF types.

```
public class MyPerson implements CustomDatum
{
  public static final String _SQL_NAME = "SCOTT.PERSON";
  public static final int _SQL_TYPECODE = OracleTypes.STRUCT;
  public static CustomDatumFactory getFactory( )                { … }
  public Datum toDatum(OracleConnection c) throws SQLException { … }
  …
  public String getFirstname()                          throws SQLException { … }
  public void setFirstname(String first_name)           throws SQLException { … }
  public String getLastname()                           throws SQLException { … }
  public void setLastname(String last_name)             throws SQLException { … }
  public java.sql.Timestamp getBirthday()               throws SQLException { … }
  public void setBirthday(java.sql.Timestamp birthday) throws SQLException { … }
}
```

Figure 4-3 - JPublisher-generated Java class for PERSON

Several observations are noteworthy about this code.

1. From our discussion of the `CustomDatum` interface components, the methods `getFactory()` and `create()`, as well as the `_SQL_TYPECODE` field should already be familiar. Object Type values are materialized as `oracle.sql.STRUCT` instances, and we specify the corresponding type code `OracleTypes.STRUCT`. We do encounter the additional field `_SQL_NAME` containing the full SQL name of the Object Type that is encapsulated here. At SQLJ translation time, this

information is necessary in order to perform type checking against the database. Additionally, the SQL type name may be required in certain situations at runtime, such as when an OUT parameter of a stored procedure or a PL/SQL block must be registered with JDBC.

2. The attributes are represented through accessor (`get`ter and `set`ter methods). This provides for more encapsulation and flexibility as compared to a representation via fields.
3. All accessors can raise an `SQLException`. This permits, for example, the Java wrapper code to flag an exception, whenever an attempt is made to retrieve a primitive (i.e. non-object) Java type, such as `int` that was read as an SQL NULL. For Java objects, we can always map Java `null` to SQL NULL.
4. The SQL types have been mapped to their JDBC counterparts: VARCHAR2 is represented as `String` and DATE as `java.sql.Timestamp`. This is the *JDBC* or *Java native* mapping. By providing a different value for JPublisher's command line option "`-mapping`", we could have requested the *Oracle native* mapping, where all SQL types will be produced as instances of corresponding `oracle.sql` classes. In our example, VARCHAR2 would be represented as `oracle.sql.CHAR` and DATE as `oracle.sql.DATE`. In this representation NULL information is preserved, since all attributes are Java objects. Furthermore, this mapping is fully information preserving, since it is based on the internal (byte-)representation of the SQL data.
5. The `MyPerson` class has a public constructor without arguments. (Not shown here.)
6. Although not shown in the code fragment above, conversions from `oracle.sql` types to JDBC types are only performed as needed. This provides a performance enhancement in many cases.
7. Users may find the different mapping styles insufficient, and might prefer to use their own customized mapping. This is also accommodated by JPublisher with a special generation mode in the type file. For example, JPublisher can generate one Java class for the SQL type PERSON, such as `PersonWrapper`, but use another Java class, say `MyPerson`, wherever a PERSON occurs in other SQL types. The user then provides the customized implementation of `MyPerson` by subclassing `PersonWrapper`.
8. Users are also able to specify customized attribute names in the type file.

It is now straightforward to use the `MyPerson` type in SQLJ programs. We also create the stored SQL function BDATE that will be called by the SQLJ code.

A new SQL function that takes a PERSON argument:

```
CREATE FUNCTION BDATE(P PERSON) RETURN DATE AS
BEGIN  RETURN P.BIRTHDAY; END;

MyPerson p;
#sql { SELECT PERS INTO :p FROM EMPS WHERE EMPNO = 1001 };
System.out.println("The name is:" + p.getFirstname() + " " + p.getLastname());
java.sql.Date d;
#sql d = { VALUES( BDATE(:p)) };
System.out.println("The birth date returned by BDATE is:" + d);
```

Figure 4-4 - Using the MyPerson wrapper in SQLJ code

In this example we use `MyPerson` in a SELECT INTO clause, and as a stored function argument. Note that whenever SQLJ performs online checking during precompilation (this is turned on by passing the "`-user`" option to the translator), it will verify that the SQL type SCOTT.PERSON is permitted wherever we are using a `MyPerson` host variable in an SQL statement.

4.3 REFs

An Oracle8 *REF* type is a persistent, strongly typed object reference defined in SQL. There are different kinds of REFs: those with system-generated globally unique Ids, scoped REFs, and user-defined constructors which allow database users to supply a primary-key instead of the ROWID into the REF structure.

Whenever JPublisher unparses an SQL Object Type and generates a Java wrapper <*JavaClass*> it also automatically generates a corresponding wrapper <*JavaClass*>`Ref` to encapsulate a strongly typed SQL REF that can reference instances of that Object Type. In the example that we discussed previously, JPublisher created an additional file `MyPersonRef.java`. The content of

this file (after removing implementation details) is shown below.

```
public class MyPersonRef implements CustomDatum
{ public static final String _SQL_BASETYPE = "SCOTT.PERSON";
  public static final int _SQL_TYPECODE = OracleTypes.REF;
  public Datum toDatum(OracleConnection c) throws SQLException { … }
  public static CustomDatumFactory getFactory() { … };

  public MyPersonRef()  { … }
  public MyPerson getValue()       throws SQLException { … }
  public void setValue(MyPerson c) throws SQLException { … }
}
```

Figure 4-5 - JPublisher-generated Java class for REF PERSON

As expected, this class implements the `CustomDatum` interface with the `toDatum()` and `getFactory()` methods and the `_SQL_TYPECODE` field. The type code for REFs is `OracleTypes.REF`. Instead of a `_SQL_NAME` field, however, we now see a `_SQL_BASETYPE` field that holds the name of the Object Type that is being referenced by this REF.

Once a `MyPersonRef` instance (representing a REF to PERSON) has been retrieved from an Oracle8 database, the value of the referenced object can be obtained with the method `getValue()`. Conversely, the referenced object can be assigned a different PERSON value with `setValue(`*new_value*`)`. Note that each of these calls is sent directly to the database. Thus, `oracle.sql.REF`s possess a purely value-based semantics.

Note that we cannot create REF's to individual table columns, such as the PERS column of the EMPS table in our example above. Thus, in the sample code below, we must first create an extent table of PERSON, before we can manipulate REFs to PERSON in SQLJ.

SQL code to prepare an extent table:

```
CREATE TABLE PERSON_EXT OF PERSON;
INSERT INTO PERSON_EXT VALUES(PERSON('Albert', 'Einstein', TO_DATE('14-MARCH-1879')));

MyPersonRef pref;
#sql { SELECT REF(p) INTO :pref FROM PERSON_EXT p WHERE p.LASTNAME = 'Einstein' };
MyPerson p = pref.getValue();
System.out.println("Birthday:" + p.getBirthday() );
p.setFirstname("Hans Albert"); p.setBirthday(new java.sql.Timestamp(04, 04, 14, 0, 0, 0));
pref.setValue(p);
java.sql.Date d;
#sql { SELECT p.BIRTHDAY INTO :d FROM PERSON_EXT p WHERE p.LASTNAME = 'Einstein' };
System.out.println("Birthday:" + d );
```

Figure 4-6 - Using MyPersonRef in SQLJ code

4.4 COLLECTIONS

Collections can be viewed as data types grouped in a certain way. They can be attributes of objects or elements of a relational table.

A *varray* is an ordered set of zero or more elements of the same type (including a user-defined Object Type). Each element has a position that uniquely identifies the element in the varray. A position is an integer ranging from 1 to the maximum declared number of elements in the varray.

A *nested table* is a one-column table that can be treated as a data type (such as an object, NUMBER, or VARCHAR2). It can be an attribute of an object or be 'nested' into a column of a relational table. A *nested cursor* is used to iterate through the rows of a nested table. This may be useful when it is too cumbersome to retrieve an entire nested table.

Analogously to the Object Type case, JPublisher can also create Java wrapper classes for varray and nested table types. Both of these SQL types are represented in Java as `oracle.sql.ARRAY`. The generated `CustomDatum` class contains the SQL type name in the `_SQL_NAME` field. Also note that varrays and nested tables cannot be referenced by SQL REFs. Consequently, JPublisher will not generate any REF wrappers for these types.

As mentioned previously, Java wrapper classes may be used in SQLJ statements wherever the corresponding SQL types occur in SQL. For brevity, we omit the SQLJ code examples for varrays and nested classes here

Nested cursors do not constitute SQL data. Rather, they represent result sets. They can be materialized in SQLJ either as a JDBC result set, or in a structured manner as named or positioned SQLJ iterator instances

5. CONCLUSION

This paper charts a gateway between SQL and Java objects. It introduces an efficient SQL data representation in Java with the `oracle.sql` package. The `CustomDatum` interface mechanism maps SQL data to user-defined Java types. We illustrate the usefulness of `CustomDatum` through an example where `Serializable` Java objects are stored in RAW database columns. We also show that the same mechanism is used for mapping SQL object types, REFs, and varray/nested table types to Java. Finally, we examine how the JPublisher tool generates Java wrappers for SQL types, and how these types are subsequently used in SQLJ programs.

SQLJ offers several advantages over JDBC. It provides much more concise syntax for embedding static SQL in Java. More importantly, it offers ahead-of-time checking of SQL syntax and semantics at translation time, rather than runtime. Type checking between SQL types and Java types is performed with an online database connection and is particularly useful when programming with structured SQL types, REFs, and varrays/nested tables. The `CustomDatum` mechanism for mapping SQL data to Java provides the necessary support for static type checking.

Additionally, all user-defined Java types that wrap SQL types can be used transparently in SQLJ programs, as if they were "built-in" JDBC types. Contrast this with corresponding Oracle JDBC programs, where users may have to use Oracle specific methods as well as providing casts from JDBC result sets and statements to their Oracle-specific implementations. All this is encapsulated in the SQL runtime provided with Oracle.

Note that you need not choose between either SQLJ or JDBC. You can use both of these fully complementary and interoperable APIs together. Their basic, efficient SQL data representation is the same. Their mechanism for mapping SQL data to user-defined Java types for Object Types, REFs, varrays/nested tables, and other SQL types is the same. The JPublisher tool is equally useful for SQLJ and JDBC programmers. Thus, user-defined data representations can be used with both, dynamic SQL statements in JDBC as well as with static statements in SQLJ.

Oracle8*i* connects Java and SQL programming in an unprecedented way. You are welcomed to travel the exciting Object road mapped out here between these two languages.

ACKNOWLEDGMENTS

This paper is based on the work of the SQLJ and JDBC teams at Oracle. I would like to thank Julie Basu, Brian Becker, Ragamayi Bhyravabhotla, Rakesh Dhoopar, Pierre Dufour, Salman Khan, Prabha Krishna, Thomas Kurian, Alan P. Thiesen, Jerry Schwarz, and Brian Wright for their contributions and comments.

REFERENCES

[1] *An Overview of SQLJ: Embedded SQL in Java,* Julie Basu. Oracle Open World 1998.

[2] *Developing Java Applications with Oracle Objects,* Prabha Krishna. Oracle Open World 1998.

[3] Oracle8*i* SQLJ, JDBC, and JPublisher user documentation. Oracle Corporation, 1998.