Tips and Tricks for SAP Java Connector (JCo) Client Programming

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's BIT525 and BIT526 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 123)

free advice often turns out to be expensive
– Terry Pratchett, The Wee Free Men, p. 50

When I teach SAP Java Connector (JCo) training classes or workshops, some of the participants have previous JCo programming experience. They may have tried the sample programs supplied with JCo or even written some JCo code themselves. Invariably, they are looking for advanced information on JCo to help them improve their applications or speed up their development projects. If you are in the same situation, then this article is for you. It contains an assortment of general recommendations, performance tips, debugging tricks, and solutions to specific client programming challenges that should be of value to all JCo developers.

If you are a Java programmer new to JCo, I recommend that you read my JCo tutorial (supplied with JCo) and then try out some of the sample programs before continuing with this article.

Take Advantage of the SAP Java IDoc Class Library

From the dawn of its time, JCo could be utilized to send and receive IDocs (Intermediate Documents). The creation or interpretation of an IDoc required a lot of programming effort, though. Not anymore. With the release of the SAP Java IDoc Class Library, you can create or interpret an IDoc with much less work. Download the product from

http://service.sap.com/connectors and try the sample programs. For a great introduction to the library, read Robert Chu's articles in the September/October 2003 issue of this publication.

Upgrade to JCo 2.1.1

When you compare the quality of different SAP products, JCo is definitely in the top 10 percent. That notwithstanding, there have been some releases that introduced "features" (a.k.a. bugs) that we could have lived without. An automatic update to the latest release is therefore dangerous. Release 2.1.1, I am happy to report, is a very stable release and I recommend that you upgrade to it sooner or later. Sooner especially if you are still using JCo 1.x, for which support has ended with 2003.

Data Type Mapping and Conversions

When you invoke an RFC-enabled Function Module (RFM) in SAP, JCo needs to translate between the ABAP and Java data types. JCo uses six different Java data types to represent the ABAP data types. **Figure 1** shows the mapping between the ABAP and Java data types used by JCo.

Corresponding to these six Java data types, JCo offers six data-type-specific access methods for fields (e.g., *getBigDecimal()*) plus a generic access method (*getValue()*). See **Figure 2** for a complete list of these seven methods.

In addition, JCo provides convenience methods that are useful if you need different data types in your application (see **Figure 3**). Some of these methods may be new even to experienced JCo developers because SAP has added them comparatively recently.

When you use an access method that does not correspond to a field's Java data type, JCo will try to convert the contents to the requested data type. A *JCO.ConversionException* will be thrown if the conversion fails.

Figure 1 *ABAP and Java Data Types*

ABAP	Description	Java	Type Constant
b	1-byte integer	int	JCO.TYPE_INT1
s	2-byte integer	int	JCO.TYPE_INT2
I	4-byte integer	int	JCO.TYPE_INT
C	Character	String	JCO.TYPE_CHAR
N	Numerical character	String	JCO.TYPE_NUM
P	Binary Coded Decimal	BigDecimal	JCO.TYPE_BCD
D	Date	Date	JCO.TYPE_DATE
T	Time	Date	JCO.TYPE_TIME
F	Float	double	JCO.TYPE_FLOAT
X	Raw data	byte[]	JCO.TYPE_BYTE
g	String (variable-length)	String	JCO.TYPE_STRING
y	Raw data (variable-length)	byte[]	JCO.TYPE_XSTRING

Figure 2 Basic Field Access Methods

public java.math.BigDecimal getBigDecimal(int index/String field_name)
public byte[] getByteArray(int index/String field_name)
public java.util.Date getDate(int index/String field_name)
public double getDouble(int index/String field_name)
public int getInt(int index/String field_name)
public java.lang.String getString(int index/String field_name)
public java.lang.Object getValue(int index/String field_name)

Figure 3 Convenience Field Access Methods

public java.math.BigInteger getBigInteger(int index/String field_name)
public java.io.InputStream getBinaryStream(int index/String field_name)
public char getChar(int index/String field_name)
public java.io.Reader getCharacterStream(int index/String field_name)
public short getShort(int index/String field_name)
public long getLong(int index/String field_name)
public java.util.Date getTime(int index/String field_name)

To change the value of a field, you utilize the *setValue()* method. This method is overloaded for the data types listed in **Figure 4**. Again, JCo will try to convert and throw an exception if you pass an unconvertible value.

For most data types, the mapping always works without requiring any special attention. There are two exceptions:

- Some BAPIs use unconventional date and/or time values. Dates like “00000000” and “99999999” as well as the time “240000” are

Figure 4 Data Types Allowed in setValue()

byte[]
char
int
long
double
short
java.lang.Object
java.lang.String

Figure 5 Treatment of Special Date and Time Values

ABAP Data Type	Contents	getString()	getDate() or getTime()
D	00000000	0000-00-00	null
D	99999999	9999-99-99	9999-12-31
T	240000	24:00:00	23:59:59

illegal in Java. In order to support BAPIs that use these values, JCo has special rules shown in **Figure 5**. As you can see, you get different results depending on whether you invoke *getString()* or *getDate()*/*getTime()*.

- Some BAPIs do not follow the rules defined for currency amounts and return incorrect amounts if the specific currency uses more or less than two decimals. Unfortunately, quite famous BAPIs like *SalesOrder.GetStatus* are amongst those in this category. For more information about this problem and how to deal with it, please read my article “Currencies and Currency Conversions in BAPI Programming” in the September/October 2001 issue of this publication.

Always Use a Fresh JCO.Function Object

Some customers have tried to optimize the performance of their applications by reusing existing *JCO.Function* objects. This is not only superfluous because *JCO.Repository* buffers the metadata for the RFMs itself, but also dangerous. If you call a function in SAP that fills a table parameter without deleting existing rows first, then more and more rows are added to this table. In other words, you get incorrect results. In order to simplify the creation of new *JCO.Function* objects, you can use a method like the one shown in **Figure 6**. Class *ARAsoftException*, which allows the chaining of exceptions, is listed in Appendix A on page 124. **Figure 7** is sample code that uses the *createFunction()* method.

Figure 6 The createFunction Method

```
public JCO.Function createFunction(String name)
                    throws ARAsoftException {
  try {
    IFunctionTemplate ft =
      mRepository.getFunctionTemplate(name.toUpperCase());
    if (ft == null)
      return null;
    return ft.getFunction();
  }
  catch (Exception ex) {
    throw new ARAsoftException(
              "Problem retrieving JCO.Function object.", ex);
  }
}
```

Figure 7 *Using the createFunction Method*

```
JCO.Function function =
  createFunction("BAPI_COMPANYCODE_GETLIST");
mConnection.execute(function);
JCO.Structure bapiReturn = function.getExportParameterList()
                                   .getStructure("RETURN");
if ( ! (bapiReturn.getString("TYPE").equals("") ||
        bapiReturn.getString("TYPE").equals("S")) ) {
  System.out.println(bapiReturn.getString("MESSAGE"));
  System.exit(0);
}
JCO.Table table =
  function.getTableParameterList().getTable("COMPANYCODE_LIST");
int records = table.getNumRows();
for (int i = 0; i < records; i++) {
  table.setRow(i);
  function = createFunction("BAPI_COMPANYCODE_GETDETAIL");
  function.getImportParameterList()
          .setValue(table.getString("COMP_CODE"),
                    "COMPANYCODEID");
  mConnection.execute(function);
  bapiReturn = function.getExportParameterList()
                       .getStructure("RETURN");
  if ( ! (bapiReturn.getString("TYPE").equals("") ||
          bapiReturn.getString("TYPE").equals("S")) ) {
    System.out.println(bapiReturn.getString("MESSAGE"));
  }
}
```

Use the Request/Response Programming Model

An RFM has three types of parameters:

- **Import:** These parameters are passed from the JCo client to the RFM. Import parameters are usually scalars (simple fields) or structures (groups of fields). In some rare cases (since 4.6C) import parameters are tables, but this feature is not used in BAPIs. Import parameters can be optional or mandatory. Scalar optional import parameters usually have a default value.
- **Export:** These parameters are passed from the RFM back to the JCo client. As for import parameters, export parameters are usually scalars or structures, and in rare cases, tables.
- **Tables:** These parameters, which can be optional or mandatory, represent ABAP internal tables, i.e., smart arrays. Table parameters are technically passed by reference. Only the documentation ("in other words, the source code"[1]) tells the developer whether the parameter is semantically to be interpreted as import, export, or both.

  When an RFM is defined as a BAPI, a table parameter can be marked as import only, export only, or both, but this setting has only informational value and is totally ignored when the BAPI is invoked as an RFM. In addition, not all BAPIs are defined correctly with regard to this setting.

[1] Tom Archer and Andrew Whitechapel, *Inside C#*, p. 145.

Figure 8 *The createRequest Method*

```
public JCO.Request createRequest(String name)
                   throws ARAsoftException {
  try {
    IFunctionTemplate ft =
      mRepository.getFunctionTemplate(name.toUpperCase());
    if (ft == null)
      return null;
    return ft.getRequest();
  }
  catch (Exception ex) {
    throw new ARAsoftException(
              "Problem retrieving JCO.Request object.", ex);
  }
}
```

Figure 9 *Using the Request/Response Programming Model*

```
JCO.Request request = createRequest("BAPI_COMPANYCODE_GETLIST");
JCO.Response response = mConnection.execute(request);
JCO.Structure bapiReturn = response.getStructure("RETURN");
if ( ! (bapiReturn.getString("TYPE").equals("") ||
        bapiReturn.getString("TYPE").equals("S")) ) {
  System.out.println(bapiReturn.getString("MESSAGE"));
  System.exit(0);
}
JCO.Table table =
  response.getTable("COMPANYCODE_LIST");
int records = table.getNumRows();
for (int i = 0; i < records; i++) {
  table.setRow(i);
  request = createRequest("BAPI_COMPANYCODE_GETDETAIL");
  request.setValue(table.getString("COMP_CODE"),
                   "COMPANYCODEID");
  response = mConnection.execute(request);
  bapiReturn = response.getStructure("RETURN");
  if ( ! (bapiReturn.getString("TYPE").equals("") ||
          bapiReturn.getString("TYPE").equals("S")) ) {
    System.out.println(bapiReturn.getString("MESSAGE"));
  }
}
```

In the standard JCo programming model, these three parameter types are represented by three objects of type *JCO.ParameterList*, accessed via the *JCO.Function* object's *getImportParameterList()*, *getExportParameterList()*, and *getTableParameterList()* methods. (See Figure 7 for sample code using these methods.)

Although most developers get used to the standard programming model after comparatively little time, some still feel uncomfortable. For these developers, JCo offers an alternative, the Request/Response programming model. Instead of explicitly creating a *JCO.Function* object, the application creates an object of type *JCO.Request*. This object then allows access to all parameters defined as Import or Tables.

To invoke the RFM, you use the *execute(JCO.Request request)* method of object type *JCO.Client*. This overloaded version of the *execute(JCO.Function function)* method returns an object of type *JCO.Response*, which allows access to all Export and Tables parameters. **Figure 8** shows a sample method (similar to the *createFunction()* method in Figure 6) that can be used to encapsulate the creation of a *JCO.Request* object. **Figure 9** contains the application code from Figure 7, rewritten to use the Request/Response programming model. There are no performance penalties for using the Request/Response model, so your choice should be based entirely on your personal preferences. Each development project should probably define a standard for this choice, so that developers have no problem when maintaining other people's code.

Use Only One Repository

The most important performance tip for applications that are clients to an SAP server is to avoid calls to said server. In general, that means caching read-only SAP information in your client application. In terms of the *JCO.Repository* class, it means to create only one repository object per SAP system that your application is connected to. The RFM metadata that the *JCO.Repository* caches is client-independent ("client" here meaning the 3-digit number that logically separates an SAP database), so one repository per SAP system is sufficient. *JCO.Repository* dynamically retrieves RFM metadata from SAP for each RFM that it could not find in its cache. Creating one repository object per user of your application, or, even worse, for each session of each user, significantly increases the number of SAP calls your application is responsible for. This will not only slow down your application, but also the SAP system itself. So the importance of creating only one repository per SAP system cannot be overestimated.

Unfortunately, JCo does not help you to accomplish this task, so you need to write a suitable repository manager class yourself. This class must keep a reference to each repository object and provide methods that allow the client program to access existing repositories or create new ones if necessary.

For more information about *JCO.Repository*, please refer to my article "Repositories in the SAP Java Connector (JCo)" in the March/April 2003 issue of this publication. An implementation of a repository manager class, *StandardRepositoryManager*, is reprinted in Appendix B on page 126 for your convenience.

Using Connection Pools

This topic warrants its own article, but I will give you the most important information here:

- Use one pool for your *JCO.Repository* and technical support components encapsulating Helpvalues, conversions between the internal and external SAP data formats, metadata retrieval, etc.
- Use one pool for the anonymous users (if any) of your application.
- Use a (small) pool for each named user of your application. Using this approach instead of explicitly created *JCO.Client* objects makes it much easier to strike the proper balance between freeing resources in SAP and avoiding too many logons to SAP.
- Return a *JCO.Client* (via a call to *releaseClient()*) that you obtained from a pool (via a call to *getClient()*) early, but not too early. It makes no sense to bracket each RFM invocation with

getClient()/*releaseClient()* if a sequence of RFMs is called successively without any user interaction or other prolonged wait periods.

Also be aware that if you use BAPIs that require an extra commit by the application, these BAPIs and the commit BAPI (*BapiService.TransactionCommit*, RFM BAPI_TRANSACTION_COMMIT) must be called in the same *getClient()*/*releaseClient()* bracket.

- Make the pool large enough so that wait situations do not occur. In older versions of JCo, the maximum value of connections in a pool, specified when the pool was created, could never be exceeded. Nowadays, there is an additional parameter that can be set using the *setMaxConnections()* method of *JCO.Pool*. When the pool is created, MaxConnections is set to the same value as MaxPoolSize. MaxConnections controls the maximum number of *JCO.Client* objects that can be obtained from the pool. MaxPoolSize controls how many *JCO.Client* objects are kept in an internal array. If a connection that is part of the internal array is returned to the pool, it will be kept open (connected to SAP) until ConnectionTimeOut (default: 10 minutes) is reached. This allows you to reuse a connection without having to go through the expensive logon process again. If a connection that is not part of the internal array is returned to the pool (this is only possible if MaxConnections is larger than MaxPoolSize), it is closed immediately.

  My recommendation is to set MaxPoolSize to a value large enough to cover any activity of your application other than absolute peaks. Make MaxConnections large enough so that the limit is never reached. An exception to this would be the small pools used for individual named users. Here a small MaxConnections is a suitable way to ensure that the same user does not have an inordinate number of sessions with the SAP system.

Use Properties for Connection Information

When you create a *JCO.Client* or *JCO.Pool* object, you have several overloaded versions of the pertinent methods to choose from. My recommendation is to avoid hard-coded values and use a *Properties* object instead. **Figure 10** contains a code snippet that

Figure 10 Using Properties to Create a JCO.Pool

```
import com.sap.mw.jco.*;

static final String POOL_NAME = "ARAsoft";

try {
  if (JCO.getClientPoolManager().getPool(POOL_NAME) == null) {
    OrderedProperties logonProperties =
      OrderedProperties.load("/logon.properties");
    JCO.addClientPool(POOL_NAME, 10, logonProperties);
  }
  JCO.Pool pool = JCO.getClientPoolManager().getPool(POOL_NAME);
}
catch (Exception ex) {
  ex.printStackTrace();
}
```

Figure 11 *Sample Contents of File logon.properties*

```
jco.client.client=001
jco.client.user=userid
jco.client.passwd=secret
jco.client.ashost=hostname
jco.client.sysnr=00
```

creates a pool object based on the information in a file (**Figure 11**). You can initialize your *Properties* object whichever way you like, but for your convenience class *OrderedProperties* is provided in Appendix C on page 130.

Inactivate Table Parameters

Some RFMs, especially BAPIs, cover a lot of application scope and thus need to have quite a few table parameters. A concrete application usually only needs a subset of all these tables. You can improve the performance of your application by inactivating those table parameters that your application does not utilize. This is accomplished by invoking the *setActive()* method, available both for the *JCO.ParameterList* and *JCO.Request* object types. **Figure 12** shows sample code that inactivates a table parameter.

Improving System Performance When Appending Multiple Table Rows

You can manipulate table parameters by deleting, inserting, or appending rows. If you need to append multiple rows, your application will run faster if you replace multiple calls to the *appendRow()* method by one call to *appendRows(int num_rows)*. An alternative approach is the *ensureBufferCapacity(int required_rows)* method, which was added in JCo 2.1.1.

Collecting Structures into a Table

A structure parameter is equivalent to a table parameter with exactly one row. Both structures and table rows are composed of fields. This is reflected by the fact that both *JCO.Structure* and *JCO.Table* are subclasses of *JCO.Record*. In some applications, we need to somehow collect the contents of multiple structure objects. An example would be an application that needs all the details for all company codes. You would first call *CompanyCode.GetList* (RFM BAPI_COMPANYCODE_GETLIST) to produce a list of all company codes and their names and then call *CompanyCode.GetDetail* (RFM BAPI_COMPANYCODE_GETDETAIL) once for each company code in order to retrieve the details provided in structure parameter COMPANYCODE_DETAIL.

JCo makes collecting all the information contained in those structures extremely easy by allowing you to collect them into a table. The first step is to create a table with the same fields (columns)

Figure 12 *Inactivating a Table Parameter*

```
function.getTableParameterList().setActive(false, "TABLE_PARAM");
```

Figure 13 *Collecting Structures into a Table*

```
import com.sap.mw.jco.*;

import de.arasoft.java.ARAsoftException;
import de.arasoft.sap.jco.BapiMessageInfo;
import de.arasoft.sap.jco.StandardRepositoryManager;

public class CompanyCodes {
  JCO.Repository mRepository;
  JCO.Client mConnection;
  static final String POOL_NAME = "ARAsoft";

  public CompanyCodes() {
    try {
      if (JCO.getClientPoolManager().getPool(POOL_NAME) == null) {
        OrderedProperties logonProperties =
          OrderedProperties.load("/logon.properties");
        JCO.addClientPool(POOL_NAME, 10, logonProperties);
      }
      JCO.Pool pool = JCO.getClientPoolManager().getPool(POOL_NAME);
      mRepository =
        StandardRepositoryManager.getSingleInstance()
                                 .getRepository(pool, true);
      mConnection = JCO.getClient(POOL_NAME);
      JCO.Function function =
        createFunction("BAPI_COMPANYCODE_GETLIST");
      mConnection.execute(function);
      BapiMessageInfo returnMessage =
        new BapiMessageInfo(function.getExportParameterList()
                                    .getStructure("RETURN"));
      if ( ! returnMessage.isBapiReturnCodeOkay() ) {

        System.out.println(returnMessage.getFormattedMessage());
        System.exit(0);
      }
      JCO.Table table =
        function.getTableParameterList().getTable("COMPANYCODE_LIST");
      function = createFunction("BAPI_COMPANYCODE_GETDETAIL");

// Create a table based on a structure
```

that the structure to be collected contains. This is accomplished by invoking the constructor of *JCO.Table*, passing the structure parameter object for COMPANYCODE_DETAIL as the only parameter (see **Figure 13** for sample code for this and subsequent operations).

Figure 13 (continued)

```
      JCO.Table infos =
        new JCO.Table(function.getExportParameterList()
                              .getStructure("COMPANYCODE_DETAIL"));
      infos.ensureBufferCapacity(table.getNumRows());

      int records = table.getNumRows();
      for (int i = 0; i < records; i++) {
        table.setRow(i);
        function = createFunction("BAPI_COMPANYCODE_GETDETAIL");
        function.getImportParameterList()
                .setValue(table.getString("COMP_CODE"),
                          "COMPANYCODEID");
        mConnection.execute(function);
        returnMessage =
          new BapiMessageInfo(function.getExportParameterList()
                                      .getStructure("RETURN"));
        if ( ! returnMessage.isBapiReturnCodeOkay(false, false,
                                             null, "FN021") ) {
          System.out.println(returnMessage.getFormattedMessage());
          System.exit(0);
        }
// Copy the data of the structure to the table
        infos.copyFrom(function.getExportParameterList()
                               .getStructure("COMPANYCODE_DETAIL"));

      }
      infos.writeHTML("c:\\infos.html");
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
    finally {
      JCO.releaseClient(mConnection);
    }
  }

  public static void main(String[] args) {
    CompanyCodes app = new CompanyCodes();
  }
}
```

Then, after each call of BAPI_COMPANYCODE_GETDETAIL, we append the structure data in the COMPANYCODE_DETAIL parameter to the table by invoking the *copyFrom()* method offered by *JCO.Table*. It is not necessary to call *appendRow()*

Figure 14 *Exception Handling*

```
JCO.Function function = this.createFunction("DDIF_FIELDINFO_GET");
try {
  function.getImportParameterList()
          .setValue("MARA", "TABNAME");
  mConnection.execute(function);
}
catch (JCO.AbapException ex) {
  if (ex.getKey().equalsIgnoreCase("NOT_FOUND")) {
    System.out.println
      ("Dictionary structure/table not found.");
    System.exit(1);
  }
  else {
    System.out.println(ex.getMessage());
    System.exit(1);
  }
}
catch (JCO.Exception ex) {
// Handle the exception
}
catch (Exception ex) {
// Handle the exception
}
```

before calling *copyFrom()*, since the latter method appends a new row automatically.

Use Proper Exception Handling

JCo uses three types of exceptions. *JCO.Exception* is the main one, and *JCO.AbapException* and *JCO.ConversionException* are its subclasses. *JCO.ConversionException* was discussed earlier. *JCO.AbapException* is thrown whenever the RFM you invoke raises an exception. All three exceptions are runtime exceptions, i.e., you are not syntactically required to catch them or define them in the method signature. This should not be interpreted as an indication that proper exception handling in your application is not necessary. **Figure 14** shows an exception handling example, in which we differentiate between the NOT_FOUND exception and other ABAP exceptions raised by DDIF_FIELDINFO_GET.

Synchronization in JCo

Sharing objects between threads is always somewhat dangerous without synchronization, so it is important to know how JCo deals with this. The rule is simple: Access to *JCO.Pool* and *JCO.Repository* objects is synchronized and nothing else! Sharing connections (*JCO.Client* objects) is specifically disallowed and will lead to an exception. Sharing other objects like *JCO.Table* objects is possible, but you need to take care of the synchronization yourself.

Create HTML to Help with Debugging

When debugging your application, you oftentimes need to check the contents of the parameters of the RFMs you invoke. Using the debugger contained in your IDE, this is a very cumbersome task. JCo

Figure 15 *The HTML Representation of a Table*

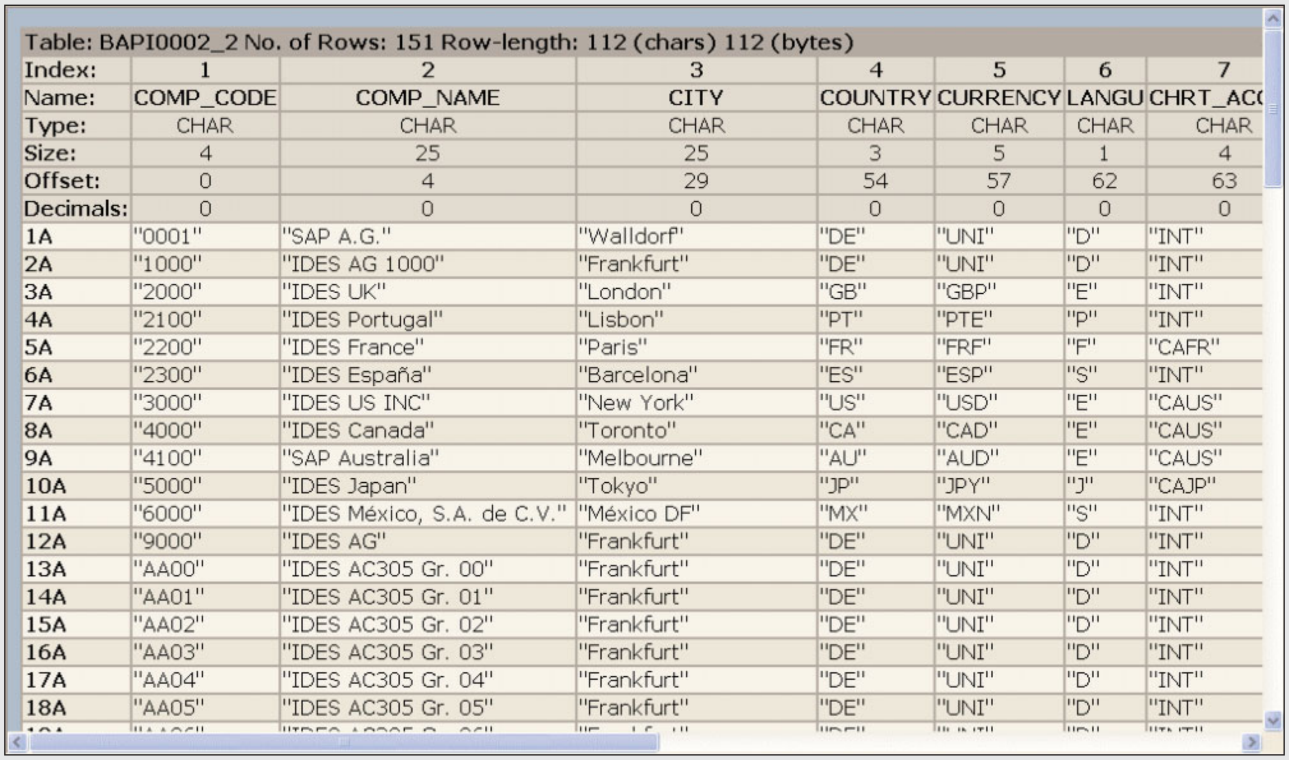


Table: BAPI0002_2 No. of Rows: 151 Row-length: 112 (chars) 112 (bytes)

Index:	1	2	3	4	5	6	7
Name:	COMP_CODE	COMP_NAME	CITY	COUNTRY	CURRENCY	LANGU	CHRT_AC
Type:	CHAR	CHAR	CHAR	CHAR	CHAR	CHAR	CHAR
Size:	4	25	25	3	5	1	4
Offset:	0	4	29	54	57	62	63
Decimals:	0	0	0	0	0	0	0
1A	"0001"	"SAP A.G."	"Walldorf"	"DE"	"UNI"	"D"	"INT"
2A	"1000"	"IDES AG 1000"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
3A	"2000"	"IDES UK"	"London"	"GB"	"GBP"	"E"	"INT"
4A	"2100"	"IDES Portugal"	"Lisbon"	"PT"	"PTE"	"P"	"INT"
5A	"2200"	"IDES France"	"Paris"	"FR"	"FRF"	"F"	"CAFR"
6A	"2300"	"IDES España"	"Barcelona"	"ES"	"ESP"	"S"	"INT"
7A	"3000"	"IDES US INC"	"New York"	"US"	"USD"	"E"	"CAUS"
8A	"4000"	"IDES Canada"	"Toronto"	"CA"	"CAD"	"E"	"CAUS"
9A	"4100"	"SAP Australia"	"Melbourne"	"AU"	"AUD"	"E"	"CAUS"
10A	"5000"	"IDES Japan"	"Tokyo"	"JP"	"JPY"	"J"	"CAJP"
11A	"6000"	"IDES México, S.A. de C.V."	"México DF"	"MX"	"MXN"	"S"	"INT"
12A	"9000"	"IDES AG"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
13A	"AA00"	"IDES AC305 Gr. 00"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
14A	"AA01"	"IDES AC305 Gr. 01"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
15A	"AA02"	"IDES AC305 Gr. 02"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
16A	"AA03"	"IDES AC305 Gr. 03"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
17A	"AA04"	"IDES AC305 Gr. 04"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"
18A	"AA05"	"IDES AC305 Gr. 05"	"Frankfurt"	"DE"	"UNI"	"D"	"INT"

offers methods that make this exercise very simple, by allowing you to create HTML files for *JCO.Record* (and hence its subclasses *JCO.ParameterList*, *JCO.Structure*, *JCO.Table*, *JCO.Request*, and *JCO.Response*) and *JCO.Function* objects. The pertinent method is *writeHTML(java.lang.String html_filename)*. If you use it for a *JCO.Table* (as shown towards the end of Figure 13), by default only the first 100 rows plus the last row of the table are written to the HTML file (see **Figure 15** for a screenshot of the beginning of the HTML file created by running the code in Figure 13). This is because a *JCO.Table* object could potentially contain millions of records, and browsers (at least Internet Explorer) have problems displaying very large HTML pages. If you need more than these 101 rows, you can change the `jco.html.table_max_rows` property by calling the *setProperty()* method of class *JCO*.

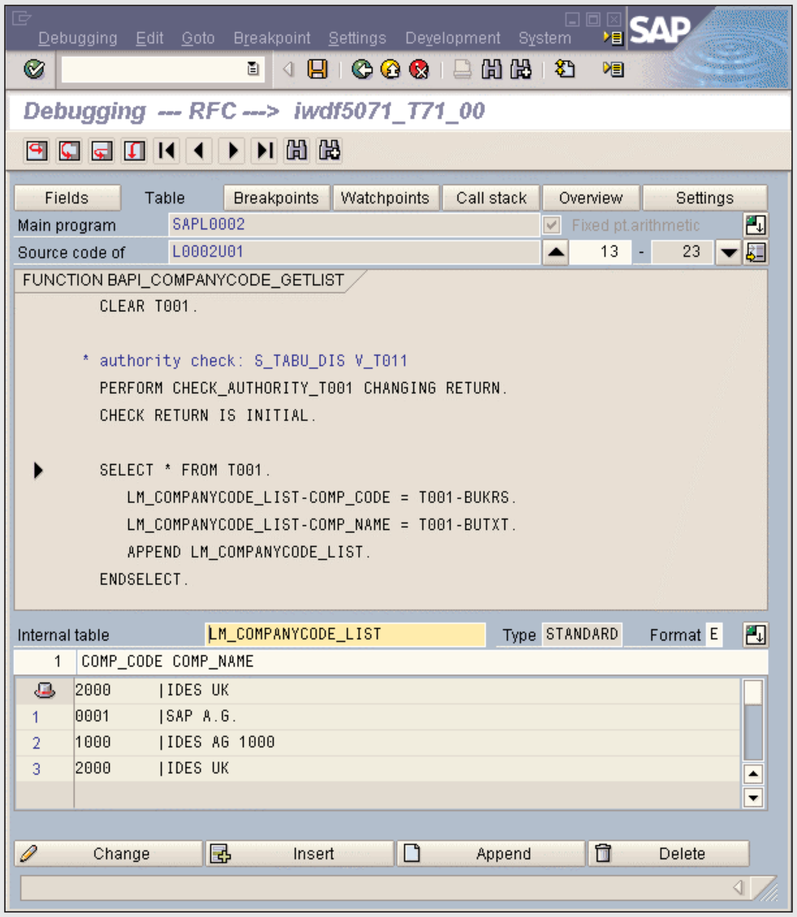
Debugging the ABAP Code

Sometimes you are quite certain that your Java code is correct and the problem to be debugged probably resides in the invoked ABAP code. How can you find out whether you are right? SAP actually allows you to debug the ABAP code if the following prerequisites are met:

- SAPGUI is installed on the machine on which the JCo client code is running.
- You invoke `setAbapDebug(true)` for the relevant *JCO.Client* or *JCO.Pool* object before the connection to SAP is established.

Then, as if by magic, the ABAP debugger will

Figure 16 *Using the ABAP Debugger*



Main program SAPL0002
Source code of L0002U01 13 - 23

```
FUNCTION BAPI_COMPANYCODE_GETLIST
        CLEAR T001.

      * authority check: S_TABU_DIS V_T011
        PERFORM CHECK_AUTHORITY_T001 CHANGING RETURN.
        CHECK RETURN IS INITIAL.

        SELECT * FROM T001.
           LM_COMPANYCODE_LIST-COMP_CODE = T001-BUKRS.
           LM_COMPANYCODE_LIST-COMP_NAME = T001-BUTXT.
           APPEND LM_COMPANYCODE_LIST.
        ENDSELECT.
```

Internal table LM_COMPANYCODE_LIST Type STANDARD Format E

```
   1 COMP_CODE COMP_NAME
     2000      |IDES UK
1    0001      |SAP A.G.
2    1000      |IDES AG 1000
3    2000      |IDES UK
```

appear. It obviously helps to be able to operate this debugger and understand ABAP source code. You may have to ask an ABAP expert for help here. See **Figure 16** for a screenshot of the ABAP debugger in action.

If the RFM that you are invoking is not in the repository cache yet, the first few calls you will see in SAP will be to metadata retrieval functions like DDIF_FIELDINFO_GET. Just continue until the RFM you are interested in is reached.

Obviously, debugging a web application this way may be a bit difficult, because you may not be allowed direct access to that server or it does not have SAPGUI installed. This is one of several reasons why I keep recommending that all SAP-related application functionality be encapsulated in classes that can be tested separately.

Combining ABAP Date and Time Fields

ABAP uses different data types for date (D) and time (T) information. In Java, both date and time

Figure 17 *Combining ABAP Date and Time Fields*

```
import com.sap.mw.jco.util.SyncDateFormat;

private static final SyncDateFormat dateISO =
  new SyncDateFormat("yyyy-MM-dd");
private static final SyncDateFormat timeISO =
  new SyncDateFormat("HH:mm:ss");
private static final SyncDateFormat dateTimeISO
  = new SyncDateFormat("yyyy-MM-ddHH:mm:ss");

public static Date combineDateAndTime(Date date, Date time) {
  try {
    return dateTimeISO.parse(dateISO.format(date) +
                             timeISO.format(time));
  }
  catch (Exception ex) { return null; }
}
public static Date combineDateAndTime(JCO.Field date, JCO.Field time) {
  try {
    return combineDateAndTime(date.getDate(), time.getDate());
  }
  catch (Exception ex) { return null; }
}
```

are contained in a *java.util.Date* object. How do you combine two ABAP fields containing date and time into one Java *Date* object? There are several possible approaches to this, but my favorite one uses the *java.text.SimpleDateFormat* class. Unfortunately, this class has a bug in at least some of Sun’s Java releases. This bug creates incorrect *Date* objects if *SimpleDateFormat* is invoked concurrently by multiple threads. The JCo developers found this bug during stress tests of JCo and decided to solve the issue by writing their own class, *com.sap.mw.jco.util.SyncDateFormat*. This class extends *SimpleDateFormat* and overwrites the buggy *format()* and *parse()* methods, making them synchronized. JCo does not provide Javadoc for *SyncDateFormat*, but that is not necessary, because it has the same features as *SimpleDateFormat*.

Figure 17 contains two *combineDateAndTime()* methods that use JCo’s *SyncDateFormat* class. The first method expects two parameters of type *Date*, the second one two *JCO.Field* objects. *JCO.Field* is a convenience class that allows you to treat fields in a uniform way, regardless of whether they are scalar parameters, fields in a structure, or fields in a row in a table.

Retrieving the SAP Date and Time

The date and time used in the SAP server is not necessarily the same as that used in the JCo client machine. SAP provides an RFM, MSS_GET_SY_DATE_TIME, that allows you to retrieve the current date and time of the SAP server.

Figure 18 *Retrieving the SAP Date and Time*

```
public static Date getSapDateAndTime(JCO.Client connection,
                                     IRepository repository)
                   throws ARAsoftException  {
  JCO.Function function = null;
  try {
    function = createFunction(repository, "MSS_GET_SY_DATE_TIME");
    if (function == null)
      throw new ARAsoftException
        ("MSS_GET_SY_DATE_TIME not found in the SAP system.");
    connection.execute(function);
    return
      combineDateAndTime(function.getExportParameterList()
                                 .getField("SAPDATE"),
                         function.getExportParameterList()
                                 .getField("SAPTIME"));
  }
  catch (ARAsoftException ax) { throw ax; }
  catch (Exception ex) {
    throw new ARAsoftException(
      "Problem invoking MSS_GET_SY_DATE_TIME.", ex);
  }
}
```

Figure 18 contains a utility method that invokes MSS_GET_SY_DATE_TIME and returns a Java *Date* object, making use of the *combineDateAndTime()* method just discussed.

Retrieving the SAP Date Format

For every R/3 user, the date format to be used in SAPGUI can be individually defined. If you want to use the same date format in your own GUI (e.g., in a browser-based application), you need to find out the date format defined in R/3. This can be accomplished by invoking RFC_GET_SAP_SYSTEM_PARAMETERS, which in addition to the date format also returns the user's language and the decimal symbol defined for the user. **Figure 19** lists a sample function for retrieving the date format from R/3. Note that RFC_GET_SAP_SYSTEM_PARAMETERS is not available in other ABAP-based SAP components like CRM.

Encapsulate the Checking of the BAPI Return Parameter

Each BAPI is supposed to have a RETURN parameter. This can be either a structure or a table. In the latter case, an empty table signifies a successful invocation of the BAPI. If the table contains rows, you need to check each of them. If a RETURN structure parameter or a row in a RETURN table parameter contains something other than "S" or an empty string in the TYPE field, then the BAPI was not completely successful. To ensure that all application programs check the RETURN parameter in a consistent fashion, it is recommended that you use a standard method like the one shown in **Figure 20**.

Figure 19 *Retrieving the SAP Date Format*

```
public static String getSapDateFormat (JCO.Client connection,
                                       IRepository repository)
                     throws Exception {
  JCO.Function function =
    repository.getFunctionTemplate("RFC_GET_SAP_SYSTEM_PARAMETERS")
             .getFunction();
  connection.execute(function);
  return
    function.getExportParameterList()
           .getString("DATE_FORMAT");
}
```

Figure 20 *The First isBapiReturnCodeOkay Method*

```
public static boolean isBapiReturnCodeOkay(JCO.Record object) {
  JCO.Structure istructure;
  JCO.Table itable;

  try {
    if (object instanceof JCO.Structure) {
      return (object.getString("TYPE").equals("") ||
              object.getString("TYPE").equals("S"));
    }
    if (object instanceof JCO.Table) {
      itable = (JCO.Table)object;
      int count = itable.getNumRows();
      if (count == 0) return true;
      boolean allOkay = true;
      for (int i = 0; i < count; i++) {
        itable.setRow(i);
        if ( ! ( itable.getString("TYPE").equals("") ||
                 itable.getString("TYPE").equals("S") ) ) {
          allOkay = false;
          break;
        }
      }
      return (allOkay);
    }
  }
  catch (Exception ex) {}
  return false;
}
```

Figure 21 *Using the First isBapiReturnCodeOkay Method*

```
JCO.Function function =
  createFunction("BAPI_COMPANYCODE_GETLIST");
mConnection.execute(function);
JCO.Structure bapiReturn = function.getExportParameterList()
                                  .getStructure("RETURN");
if ( ! isBapiReturnCodeOkay(bapiReturn) ) {
  System.out.println(bapiReturn.getString("MESSAGE"));
  System.exit(0);
}
```

Figure 22 *The Second isBapiReturnCodeOkay Method*

```
public static boolean isBapiReturnCodeOkay(JCO.Function function) {
  JCO.ParameterList exports = function.getExportParameterList();
  if (exports != null && exports.hasField("RETURN")) {
    return isBapiReturnCodeOkay(exports.getStructure("RETURN"));
  } else {
    JCO.ParameterList tables = function.getTableParameterList();
    if (tables != null && tables.hasField("RETURN")) {
      return isBapiReturnCodeOkay(tables.getTable("RETURN"));
    } else {
      return false;
    }
  }
}
```

This method defines a parameter of type *JCO.Record*, the superclass of both *JCO.Structure* and *JCO.Table*.

Figure 21 shows sample code using the *isBapiReturnCodeOkay()* method.

One disadvantage to the *isBapiReturnCodeOkay()* method from Figure 20 is that the application program must pass the RETURN parameter. If you use the standard JCo programming model, that means that the application must either access the export parameter list or the table parameter list, depending on how the BAPI has defined the RETURN parameter.

This slight inconvenience can be removed by providing an overloaded version of the *isBapiReturnCodeOkay()* method that accepts an object of type *JCO.Function* (**Figure 22**). This new version finds out for itself whether the RETURN parameter is a structure or a table and then calls our first *isBapiReturnCodeOkay()* method with the correct parameter.

Figure 23 *Using the Second isBapiReturnCodeOkay Method*

```
JCO.Function function =
  createFunction("BAPI_COMPANYCODE_GETLIST");
mConnection.execute(function);
if ( ! isBapiReturnCodeOkay(function) ) {
  System.exit(0);
}
```

Figure 23 shows sample code using the second method.

And What About JCo Server Programming?

This article has dealt exclusively with JCo client programming, which is what most projects require. For an introduction to JCo server programming, please read my article "Server Programming with the SAP Java Connector (JCo)" in the September/October 2003 issue of this publication.

action in an absence of information
is wasted effort
– Robert Asprin, Hit or Myth, p. 156

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

Appendix A: Class ARAsoftException

```
package de.arasoft.java;

/*
 * Copyright (c) 2001 ARAsoft GmbH
 * All Rights Reserved.
 */

/**
 * Exception class
 *
 * @author ARAsoft GmbH
 * @version 1.0
 * @since 1.0
 */

public class ARAsoftException extends Exception {

  private Exception mOriginalException = null;
  private final static Copyright copyright = new Copyright();

/**
 * Constructor.
 */
  public ARAsoftException() {
    super();
  }

/**
 * Constructor to be used if we wrap another exception.
 * @param originalException The original exception
 */
  public ARAsoftException(java.lang.Exception originalException) {
    super(originalException.toString());
    mOriginalException = originalException;
  }

/**
 * Constructor with a message string.
 * @param s Exception string
 */
 public ARAsoftException(String s) {
    super(s);
  }
/**
 * Constructor to be used if we wrap another exception and have an
 * additional message string.
 * @param s Exception string
 * @param originalException The original exception
 */
  public ARAsoftException(String s, Exception originalException) {
    super(s);
    mOriginalException = originalException;
  }

/**
 * Returns the original exception.
 * @return The original exception.
 */
  public Exception getOriginalException() {
    return mOriginalException;
  }

/**
 * Returns the exception message.
 * @return The exception message.
 */
  public String getMessage() {
    if (mOriginalException == null)
      return super.getMessage();
    return super.getMessage() + '\n' +
           "Original exception:" + '\n' +
           mOriginalException.toString();
  }
}
```

Appendix B: Class StandardRepositoryManager

```
package de.arasoft.sap.jco;

import java.util.TreeMap;
import com.sap.mw.jco.*;
import de.arasoft.java.ARAsoftException;

/*
 * Copyright (c) 2002 ARAsoft GmbH
 * All Rights Reserved.
 */

/**
 * A singleton object that manages JCO.Repository objects.
 *
 * @author ARAsoft GmbH
 * @version 2.5
 * @since 2.5
 */

public class StandardRepositoryManager {

  static private StandardRepositoryManager repositoryManager = null;
  static private TreeMap items = null;

  protected StandardRepositoryManager() {
    items = new TreeMap();
  }

/**
 * Returns the singleton instance of this class.
 * @return The singleton instance.
 */
  static public synchronized StandardRepositoryManager
                            getSingleInstance() {
    if ( repositoryManager == null )
      repositoryManager = new StandardRepositoryManager();
    return repositoryManager;
  }

/**
 * Creates a JCO.Repository object for the SAP system to which the pool
 * is connected.
 * Throws an exception if a repository for this system already exists.
```

```
   * @return The created repository object.
   * @param pool The JCO.Pool object.
   */
    public synchronized JCO.Repository createRepository(JCO.Pool pool)
           throws ARAsoftException {
      JCO.Client client = null;
      try {
        client = JCO.getClient(pool.getName());
        String name = client.getAttributes().getSystemID();
        JCO.releaseClient(client);
        client = null;
        if ( items.containsKey(name) )
          throw new ARAsoftException
          ("A repository for system '" + name + "' already exists.");
        JCO.Repository repository =
          new JCO.Repository(name, pool.getName());
        items.put(name, repository);
        return repository;
      }
      catch (Exception ex) {
        throw new ARAsoftException(ex);
      }
      finally {
        if ( client != null ) {
          JCO.releaseClient(client);
        }
      }
    }

  /**
   * Checks whether a JCO.Repository object for the specified SAP system
   * already exists.
   * @return Does a repository for the specified SAP system exist?
   * @param systemId The system ID of the SAP system.
   */
    public boolean existsRepository(String systemId) {
      JCO.Repository repository = (JCO.Repository) items.get(systemId);
      return ( repository != null );
    }

  /**
   * Checks whether a JCO.Repository object for the SAP system to which
   * the pool is connected already exists.
   * @return Does a repository for this system exist?
   * @param pool The JCO.Pool object.
   */
    public boolean existsRepository(JCO.Pool pool)
                   throws ARAsoftException {
      JCO.Client client = null;
   try {
```

(continued on next page)

(continued from previous page)

```
        client = JCO.getClient(pool.getName());
        String name = client.getAttributes().getSystemID();
        JCO.releaseClient(client);
        client = null;
        return this.existsRepository(name);
      }
      catch (Exception ex) {
        throw new ARAsoftException(ex);
      }
      finally {
        if ( client != null ) {
          JCO.releaseClient(client);
        }
      }
    }

  /**
   * Returns the JCO.Repository object for the SAP system to which the
   * pool is connected.
   * Throws an exception if no repository exists.
   * @return The repository object.
   * @param pool The JCO.Pool object.
   */
    public synchronized JCO.Repository getRepository(JCO.Pool pool)
           throws ARAsoftException {
      return this.getRepository(pool, false);
    }

  /**
   * Returns the JCO.Repository object for the SAP system to which the
   * pool is connected. If no repository exists and
   * <code>createIfItDoesNotExist</code>
   * is <code>true</code>, a new repository is created,
   * otherwise an exception is thrown.
   * @return The repository object.
   * @param pool The JCO.Pool object.
   * @param createIfItDoesNotExist Should a new repository be created
   *        if none exists?
   */
    public synchronized JCO.Repository getRepository
           (JCO.Pool pool, boolean createIfItDoesNotExist)
          throws ARAsoftException {
      JCO.Client client = null;
      try {
        client = JCO.getClient(pool.getName());
        String name = client.getAttributes().getSystemID();
        JCO.releaseClient(client);
        client = null;
        try {
          return this.getRepository(name);
      }
      catch (ARAsoftException ax) {
        if ( createIfItDoesNotExist ) {
          return this.createRepository(pool);
        } else {
          throw ax;
        }
      }
    }
    catch (Exception ex) {
      throw new ARAsoftException(ex);
    }
    finally {
      if ( client != null ) {
        JCO.releaseClient(client);
      }
    }
  }

/**
 * Returns the JCO.Repository object for the specified SAP system.
 * If no repository exists an exception is thrown.
 * @return The repository object.
 * @param systemId The system ID of the SAP system.
 */
  public synchronized JCO.Repository getRepository(String systemId)
         throws ARAsoftException {
    JCO.Repository repository = (JCO.Repository) items.get(systemId);
    if ( repository == null )
      throw new ARAsoftException
                ("No repository exists for system '"
                 + systemId + "'.");
    return repository;
  }

/**
 * Removes the specified JCO.Repository object.
 * @param repository The repository to be removed.
 */
  public synchronized void removeRepository(JCO.Repository repository)
 {
    String name = repository.getName();
    if ( items.containsValue(repository) ) {
      items.remove(name);
    }
  }
}
```

Appendix C: Class OrderedProperties

```
import java.util.*;
import java.io.*;

public class OrderedProperties extends java.util.Properties {
  ArrayList orderedKeys = new ArrayList();

  public OrderedProperties() {
    super();
  }
  public OrderedProperties(java.util.Properties defaults) {
    super(defaults);
  }

  public synchronized Iterator getKeysIterator() {
    return orderedKeys.iterator();
  }

  public static OrderedProperties load(String name)
                                throws Exception {
    OrderedProperties props = null;
    java.io.InputStream is =
      OrderedProperties.class.getResourceAsStream(name);
    props = new OrderedProperties();
    if (is != null) {
      props.load(is);
      return props;
    }
    else {
      throw new IOException("Properties could not be loaded.");
    }
  }

  public synchronized Object put(Object key, Object value) {
    Object obj = super.put(key, value);
    orderedKeys.add(key);
    return obj;
  }

  public synchronized Object remove(Object key) {
    Object obj = super.remove(key);
    orderedKeys.remove(key);
    return obj;
  }
}
```