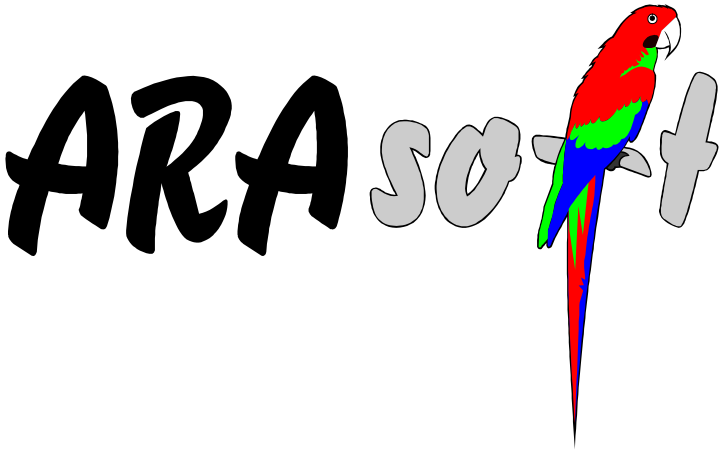


Thomas G. Schuessler, www.ARAsoft.de: Developing Applications with the "SAP Java Connector" (JCo)

Sponsor programme and close down.
James Joyce: Finnegans Wake, p. 531.27

The more complex the system and the more expert the users,
the more their technical conversation sounds like the plot of a soap opera.
Steven Pinker: How the Mind Works, p. 78

Version 0.8.1

1. Introduction

1.1. New in this Version (0.8.1)

I have added Appendix A-8 which contains a full-blown, annotated example of how to create a sales order.

1.2. Disclaimer

This text is a preliminary, incomplete version of a book that I will publish on programming with JCo. The complete book will have much more information. No guarantee is given for the technical accuracy of any information in this text. We are not responsible for any damage resulting from the use of the presented information or running the associated sample programs.

1.3. What is JCo?

SAP's Java middleware, the SAP Java Connector (JCo) allows SAP customers and partners to easily build SAP-enabled components and applications in Java. JCo supports both inbound (Java calls ABAP) and outbound (ABAP calls Java) calls in desktop and (web) server applications.

1.4. Goal and Scope of This Text

This text wants to teach you how to build client applications using JCo. You are supposed to already know Java. Some understanding of SAP would also be helpful, but you should be able to get by even if you do not know much about SAP yet. We will cover all relevant aspects of client programming (where Java calls SAP). The final version of this text will also introduce outbound (server) programming.
This text does not make the study of the JCo documentation superfluous. It is probably a good idea to look at the Javadocs for the various classes in a separate window while you work through this text.
To try out the accompanying sample programs, you need only[1] a JDK and JCo installed, but a decent development environment (like JBuilder[2]) would make your life a lot easier.

1.5. Organization of This Text

In order not to confuse beginners, advanced material is presented in **Appendix A**. The text will tell you which chapter in this appendix will give you more in-depth information about a topic. **Appendix B** is a listing of the sample programs associated with this text.

[1] Unless specifically stated otherwise.
[2] We now offer a nice proxy generator integrated into JBuilder, see Appendix A-7.

2. SAP Remote Function Call (RFC)

Before you can start with JCo programming, you need to understand the SAP architecture to a certain extent, particularly how you can invoke SAP functionality from the outside (client programming) and how ABAP applications can invoke your components (server programming).
The basis for all communication between SAP and external components (as well as for most communication between SAP components) is the Remote Function Call (RFC) protocol. RFC comes in three flavors.
Most client programs want to use regular, Synchronous RFC (sRFC), but SAP also supports Transactional RFC (tRFC) and Queued RFC (qRFC). tRFC is used mainly to transfer ALE Intermediate Documents (IDocs). Currently, this text covers only sRFC, but JCo also supports tRFC and qRFC.

2.1. BAPIs and Other RFMs

ABAP Function Modules can only be called from an external client if they are marked as RFC-enabled. R/3 contains several thousands of such RFC-enabled Function Modules (RFMs). Amongst them are the BAPIs. BAPIs are RFMs that follow additional rules and are defined as object type methods in SAP's Business Object Repository (BOR). Use transaction codes `BAPI` and `SE37` to investigate the metadata of the BAPIs and other RFMs in SAP. If you do not have access to an SAP system, or you want to look up interfaces in a different release, use the SAP Interface Repository (`http://ifr.sap.com`).
An RFM can be released for customer use or not.[3] Most BAPIs are released RFMs, only very new ones may be unreleased at first so that SAP and the customers can test them for a while before they are released in a subsequent release. On the other hand, there are quite a few extremely useful RFMs that are not officially released for customer use. Many of these RFMs are not documented (or only in German), which makes it harder to figure out exactly how they work. Additionally, SAP has reserved the right to make incompatible changes to unreleased RFMs. Using these directly in your applications could thus result in a huge maintenance effort. Hence all access to unreleased RFMs must be done through components in order to limit maintenance to this one component as opposed to potentially many individual applications.

2.2. RFC-enabled Function Modules (RFMs)

RFMs can have three types of parameters, import (the client sends these to the RFM), export (RFM sends these back to the client), and tables (bi-directional). Import and export parameters can be simple fields (a.k.a. scalars) or structures.[4] A structure is an ordered set of fields.

[3] Actually, there are three states: not released, released within SAP, released for customer use. For people outside of SAP, the two former categories can be subsumed under unreleased.
[4] Since some time SAP now also allows complex structures and tables as parameters. The BAPIs do not use this capability, as far as I know (nothing is ever really certain in BAPI land). JCo supports complex parameters, but this text ignores them for now.

A table parameter has one or more columns (fields) and contains zero or more rows. Import and table parameters can be mandatory or optional, export parameters are always optional. You can check these attributes in SAP or even at runtime.
Some people assume that table parameters are somehow magically linked to the database tables in SAP. That is not the case. If you change a table that an RFM call returned, nothing happens in SAP. A table is really just a form of glorified array (glorified because we have metadata for the columns). Changes in SAP will only happen if you call an RFM that uses a table parameter you pass to it in order to update the database.
RFMs can also define ABAP exceptions. An ABAP exception is a string (e.g. `NOT_FOUND`) with an associated language-dependent message text. We will discuss exception handling below.

2.3. The SAP Data Dictionary

In RFC programming, eventually we always deal with fields. These fields can be scalar parameters themselves or contained in a structure or table row. Most, but not all, fields used in RFMs are based on Data Dictionary (DD) definitions. Each field in the DD is based on a Data Element (which in turn is based on a Domain) or a built-in DD data type. The Domain defines the basic technical attributes (data type, length, conversion routine, check table, etc.), whereas the Data Element contains the more semantical information (texts and documentation).
There is a lot of metadata available for each field in SAP. I will now introduce the most important attributes and how they affect your applications (more details can be found in the next section):

- Data type, length, number of decimals: These are essential for dealing with fields correctly and building nice user interfaces that are aware of the field attributes. JCo makes these attributes available to the client program.
- Check table: Many fields contain codes. End-users do not want to deal with them, at least not without the associated descriptions. When you call a BAPI and display the returned country code, the users will not be happy. They need the name of the country instead of – or together with – the code.
  If users have to enter codes, they do not want to guess; they want a list of available codes (Helpvalues) with their descriptions.
- Fixed values: This is similar to check tables, but the information about the codes is stored in the Domain, not in a separate table.
- Conversion exit: Many fields use conversion exits in SAPGUI to translate between the internal and external representations of data. Most BAPIs return and expect the internal format, which makes little to no sense to your users.
- Texts and documentation: SAP stores multiple texts per field and also extended documentation in many cases. This documentation is available in all installed languages and therefore an easy way to provide multi-lingual capabilities in your applications.
- Mixed case support: Many text fields in SAP are uppercase only. If users exercise care in entering data in mixed case, they will not be too happy to discover later that everything was capitalized in SAP. Your user interface should exploit the mixed case attribute to indicate to the users which fields are uppercase only.

The ARAsoft JCo Extension Library allows easy access to all the required metadata that JCo does not expose. The library also offers services that solve all the practical problems associated with the metadata, including Helpvalues, conversions, text retrieval. See below for details.

2.4. BAPI Specifics

As stated above, a BAPI is an RFM that follows additional rules (defined in the SAP BAPI Programming Guide[5]) and is defined as a method of an object type in the BOR. An example for a BAPI, as defined in the BOR, is *SalesOrder.CreateFromDat1*. The actual RFM implementing this BAPI is `BAPI_SALESORDER_CREATEFROMDAT2`. When comparing the metadata shown by the BAPI Explorer (transaction code `BAPI`) and those in the Function Builder (transaction code `SE37`), you will notice that there can be discrepancies like different parameter names. When writing your actual application code, you need to use the names presented by the Function Builder. It is best to use the BAPI Explorer just as a convenient starting point to find suitable BAPIs and then review the actual metadata of the RFM in the Function Builder (unless you use proxies, see below). The BOR presents certain attributes not available in the Function Builder:

- Is the BAPI obsolete? When SAP wants to change a BAPI in a way that would be incompatible with the existing version, they create a new BAPI instead, e.g. *Create1* would be the new version of *Create*. The old BAPI becomes obsolete. An obsolete BAPI is guaranteed to exist and work in the release in which it is marked as obsolete and the subsequent functional release. For example, a BAPI made obsolete in 4.0A would still be valid in 4.5B (the maintenance release for the functional release 4.5A), but might disappear in 4.6A.
- Is the BAPI released? Checking this is of utmost importance. SAP sometimes creates new BAPIs without releasing them yet. Only released BAPIs are guaranteed to be upward-compatible, though, so you should only use unreleased BAPIs if you have no other choice.
- Does the BAPI pop up SAPGUI dialogs? These BAPIs were built mainly for the communication between SAP components, they rarely make any sense in a Java application. SAPGUIs are difficult to pop up from a web application running in a browser…

If you want to use BAPIs in a more object-oriented way, you need to utilize proxy classes. These proxies have the following advantages:

- A business object type in the BOR is represented by one Java class, the BAPIs are methods of the class.
- Instead of the somewhat ugly function and parameter names in the SAP Function Builder you use the nice names in the BOR.
- You can exploit additional BOR metadata. One example is that the BOR knows which table parameters are import, which are export, and which are import and export. The Function Builder has no notion of this.

[5] Not all BAPIs follow all the rules, though. This can lead to a lot of confusion and frustration for developers. This texts points out some of the deviations and their consequences.

- You can use an easy request/response programming model. Instead of dealing with three parameter categories, viz. import, export, and tables, you simply have request and response parameters.
- You can use the Code Completion / Code Insight / Intellis*nse features of your Java IDE and do not have to type hard-coded function and field names like with native JCo.
- You can generate proxies that optionally exclude those BAPIs that are obsolete, not released yet, or that pop up SAPGUI dialogs.

All of these advantages do not imply that you have to use proxies. Many people (including myself) are perfectly happy with native JCo. If on the other hand, you like the idea of using proxies, please turn to Appendix A-7.
When using BAPIs in your applications, you must also be aware of some idiosyncrasies (some of them were introduced briefly in the section dealing with the SAP Data Dictionary). They are discussed in the following sections.

2.4.1. Conversions

BAPIs mostly use the internal representation of codes. One of the more famous examples is units of measurement: A SAPGUI user logged on in English will type `PC` (piece) as a unit while the BAPIs use the internal (German) representation `ST`. Another example is customer numbers. A customer number can contain letters, but if it consists of digits only, it is internally stored with leading zeroes. The user wants to neither see nor have to enter these leading zeroes.
SAPGUI automatically performs the necessary conversions so that its users always see the external representation. This is possible since for each Domain a conversion routine (sometimes called a conversion exit) can be defined if appropriate. This exit is called both inbound (the user enters data to be passed to the application) and outbound (the application returns data to be displayed in SAPGUI). Data originating in the application is converted from the internal to the external format; data entered by the user is first validated and then converted from the external to the internal format.
The fact that, in SAPGUI, the conversion exits are always called automatically has been a source of confusion for many developers who wanted to try out a BAPI in SAP's Function Builder (transaction code `SE37`) in order to learn more about the BAPI's parameters. If you run the *SalesOrder.CreateFromDat2* (RFM `BAPI_SALESORDER_CREATEFROMDAT2`) BAPI inside SAP, for example, and enter the document (sales order) type as `OR` (for a standard order), all works well. If, on the other hand, you use the same code from outside of SAP, you will receive an error message telling you that you have used an invalid code.
This is due to the fact that even the very technical test environment of the Function Builder uses the conversion exits. But the BAPI itself does not invoke them. Many developers in this situation end up hard-coding the German equivalent of `OR` (`TA`) in their programs. That may be acceptable for a quick-and-dirty demo program, but software for production use should avoid hard-coding constants that are subject to SAP customization. Also, conversions are required for many other parameters, and it would definitely be better to have a generic solution. The ARAsoft JCo Extension Library (see below) automates the necessary conversions for you, but you can use the conversion BAPIs

found on object type BapiService if you want to build your own component dealing with conversions.

2.4.2. Helpvalues

The BAPIs return lots of codes. When you invoke *Customer.GetDetail*, for example, you get back, amongst other information, the code of the country in which this customer resides. Users do not really remember what all the codes stand for, though. They want text (the code's description) in addition to, or even instead of, the code. Along the same lines, when a user has to enter a code (e.g., a country code when adding a new customer), it would be nice if we offered a drop-down combo box with all available choices (showing the text by itself or the code and the text) instead of a plain text box where the user has to guess the correct code.
In some cases, SAP offers suitable BAPIs to retrieve the required information. *CompanyCode.GetList* and *GetDetail* offer a list of all company codes with associated descriptions and detailed information for one company code, respectively.
In most cases, though, we are not that lucky. There is no *Country* object type, for instance. The *Helpvalues* object type provides BAPIs that can help us to deal with those entities for which there is no object type in SAP. To figure out whether there is *Helpvalues* support for a given entity, you must verify whether a check table or a fixed values list is defined for the field (`DDIF_FIELDINFO_GET` allows you to do this at runtime, but you can also look it up in the DD when you build your application).
Unfortunately, the *Helpvalues* BAPIs are hard to use and, to make matters worse, they return different results in different SAP releases. The BAPIs are also relatively slow, so it is imperative that we make as few calls to them as possible and cache the results. To complicate matters further, some of the entities form hierarchies. Countries, for example, contain regions (states, provinces, *Kantone, Bundesländer*). There are even multi-level hierarchies (e.g., Sales Organizations containing Distribution Channels containing Divisions containing Sales Offices). You clearly need a component to deal with all these issues. If you want to avoid having to build the required component, take a look at the ARAsoft JCo Extension Library (see below) which takes care of all the necessary processing.

2.4.3. BAPI return messages

All BAPIs are supposed to use a `RETURN` parameter instead of throwing ABAP exceptions.[6] After each BAPI call, you should[7] check the message that came back from SAP to see whether the BAPI call was successful. You have to be very careful here, though, since different BAPIs use different structures with different field names for the `RETURN` parameter.
The ARAsoft JCo Extension Library contains a class (*BapiMessageInfo*) that hides these inconsistencies from the developer and also allows easy access to the documentation for a BAPI message.

[6] Some BAPIs throw exceptions nevertheless (cf. the footnote above about BAPIs not following the rules). Always check in the SAP Function Builder whether a given BAPI has defined exceptions.
[7] Read: *must*.

2.4.4. Currency amounts

Internally, SAP uses data type CURR to represent currency amounts. CURR is a packed number (or Binary Coded Decimal, if you prefer) with two decimals. How are currencies that use a different number of decimals stored? In order to be able to store large amounts in currency fields, SAP shifts the amount so that the last digit of the value is stored in the second decimal position. Two Japanese Yen, for example, are stored as 0.02, which is wrong by a factor of 100. SAP internally knows how to handle this and converts the amounts as required before displaying them. In order to avoid that extra effort for BAPI client programmers, SAP decided not to use data type CURR in BAPIs. Instead, the BAPI Programming Guide states: "All parameters and fields for currency amounts must use the domain BAPICURR with the data element BAPICURR_D or BAPICUREXT with the data element BAPICUREXT." Not all BAPIs follow the rules, though. Always double-check that the BAPI currency amount fields you use in your applications follow the rules. Otherwise, you need to correct the value yourself, or let the ARAsoft JCo Extension Library do it for you (method *getCorrectAmount()* of class *JCoRepository*).

2.4.5. Delegation

The Business Object Repository supports a concept called Delegation. This is used when you subclass an object type and overwrite one or more of the BAPIs to enforce your own business rules. If an SAP object type is delegated to one of its subclasses, you should always call the RFM defined in the subclass instead of the standard SAP one.
If your company uses Delegation, or you want to sell your product to a customer who does, you should always determine the name of the correct RFM by using some kind of properties file or looking up the correct name dynamically using metadata access services. In order to avoid having to build your own metadata component, you could use the ARAsoft Java BAPI ObjectFactory that comes with the ARAsoft JCo Extension Library.

3. JCo Overview

JCo is a high-performance, JNI-based middleware for SAP's Remote Function Call (RFC) protocol. JCo allows you to build both client and server applications. JCo works with any SAP system starting with 3.1H. JCo is available for Windows 32, Linux, and other platforms.
JCo's ability to use connection pooling makes it an ideal choice for web server applications that are clients of an SAP system, but you can also develop desktop applications.

3.1. Downloading JCo

You need to be an SAP customer or partner (with a valid SAPnet (OSS) userid) to download JCo. Go to `http://service.sap.com/connectors` and follow the download instructions. You will also find the complete list of supported platforms here.

3.2. Installing JCo

3.2.1. JCo 1.1

Make sure that you are always using the latest JCo 1.1 version, but at least 1.1.03.
The following instructions apply to Windows 32 platforms. The instructions for installing JCo on other platforms are contained in the appropriate JCo download.

1. Create a directory, e.g., `c:\JCo`, and unzip the JCo zip file to that directory, keeping the directory structure intact.
2. Copy the **`librfc32.dll`** file from your JCo root directory to C:\WINNT\SYSTEM32 unless the version you already have there is newer than the one supplied with JCo.
3. Make sure that **`jCO.jar`** (found in your JCo root directory) is in the class path of any project that wants to use JCo.

For production deployment, the following files from the JCo zip file are required:

- `jCO.jar`
- `librfc32.dll`
- `jRFC11.dll` (if you are using JDK 1.1)
- `jRFC12.dll` (if you are using Java 2)

What else do you get in the downloaded zip file?

- The `docs` directory contains the **Javadocs** for JCo. Start with the `index.html` file.
- The `demo` directory contains some **sample programs**, including, but not limited to, the samples discussed in this text.

3.2.2. JCo 2.0

JCo 2.0 is available with SAP Basis Release 6.20. JCo 2.0 does not support JDK 1.1 anymore, but requires Java 2 (JDK 1.2 and later). If you still need to use JDK 1.1, please keep using JCo 1.1 (1.1.03 or later).

The following instructions apply to Windows 32 platforms. The instructions for installing JCo on other platforms are contained in the appropriate JCo download.

4. Create a directory, e.g., `c:\JCo`, and unzip the JCo zip file to that directory, keeping the directory structure intact.
5. Copy the **`librfc32.dll`** file from your JCo root directory to C:\WINNT\SYSTEM32 unless the version you already have there is newer than the one supplied with JCo.
6. Make sure that **`sapjco.jar`** (found in your JCo root directory) is in the class path of any project that wants to use JCo.

For production deployment, the following files from the JCo zip file are required:

- `sapjco.jar`
- `librfc32.dll`
- `sapjcorfc.dll`

What else do you get in the downloaded zip file?

- The `docs` directory contains the **Javadocs** for JCo. Start with the `index.html` file.
- The `demo` directory contains some **sample programs**, including, but not limited to, the samples discussed in this text.

4. Connecting to SAP

JCo supports two programming models for connecting to SAP: **direct connections**, which you create and hold on to as long as you want, and **connection pools**, from which you take a connection when you need it and to which you return it as soon as possible so that others can use it. These two models can be combined in one application.
If you are building web server applications, you definitely want to use connection pools[8], but they can also be used in desktop applications.

4.1. Direct Connections

In our first sample program, *TutorialConnect1*, we want to connect to SAP, display some connection attributes, and finally disconnect. The complete code for this, as well as for all other sample programs, is listed in Appendix B.

4.1.1. New classes introduced

JCO[9]	The main class of the SAP Java Connector. Offers many useful static methods.
JCO.Client	Represents a connection to SAP.
JCO.Attributes	Attributes of a connection, e.g., the release of the SAP system.

4.1.2. Import statements

Any program using JCo should contain the following import statement:

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

Otherwise, you have to fully qualify each JCo class and interface which is very inconvenient.

4.1.3. Defining a connection variable

```
JCO.Client mConnection;
```

A connection with SAP is handled by an object of class *JCO.Client*. Since the term *client* means a logical partition of an SAP system (and has to be entered when you log on to SAP, for example), this text calls an object of class *JCO.Client* a *connection*.

[8] Details about how to combine connection pools and direct connections in the same application are discussed in Appendix A-1.
[9] SAP abbreviates the SAP Java Connector as JCo. The main class of the product is called JCO. The name of the jar file in JCo 1.1.x is jCO.jar. Don't ask.

4.1.4. Creating a JCO.Client object

```
// Change the logon information to your own system/user
mConnection =
  JCO.createClient("001", // SAP client
    "<userid>",           // userid
    "****",               // password
    "EN",                 // language (null for the default language)
    "<hostname>",         // application server host name
    "00");                // system number
```

You do not use a constructor to instantiate *JCO.Client* objects. Instead, you use the *createClient()* method of class *JCO*. There are several overloaded versions of this method to support:

- Connections to a specific application server (as in the above example),
- Connections to a load balancing server group,
- Connections based on the information in a *java.util.Properties* object. This is especially useful in order to avoid hard-coded system/user information in the Java code.

Several other versions are described in the Javadocs[10].

4.1.5. Opening the connection

Creating the *JCO.Client* object does not connect to SAP, but a subsequent call to *connect()* will accomplish this:

```
try {
  mConnection.connect();
}
catch (Exception ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

4.1.6. Calling a function

Now we are ready to call functions in SAP. Since we need to learn a few more things before we can call actual application functions, in this sample program we just print out the RFC attributes for our connection.

```
System.out.println(mConnection.getAttributes());
```

See the Javadoc for *JCO.Attributes* for a discussion of the individual properties.

[10] If you must use a router string to access your SAP system, the router string is specified together with the host name in the following format: `"/H/<saprouter>/H/<hostname>"`

4.1.7. Closing the connection

After we have accomplished whatever it was we wanted to do (call one BAPI or a few hundred), eventually we want to disconnect again:

```
mConnection.disconnect();
```

As opposed to using connection pools (see below), where you want to return the connection to the pool as soon as possible, for direct connections it is not a good idea to connect to SAP, call one function, and disconnect again for every function call. There is some overhead involved in logging on to SAP and therefore we should stay connected until we are finished or until we know that there will be no further activity for quite some time.

4.1.8. Sample output

The output from running *TutorialConnect1* should look similar to this:

```
DEST:                <unknown>
OWN_HOST:            arasoft1
PARTNER_HOST:        hostname
SYSTNR:              00
SYSID:               XYZ
CLIENT:              400
USER:                TGS
LANGUAGE:
ISO_LANGUAGE:
OWN_CODEPAGE:        1100
OWN_CHARSET:         ISO8859_1
OWN_ENCODING:        ISO-8859-1
OWN_BYTES_PER_CHAR: 1
PARTNER_CODEPAGE:    1100
OWN_REL:             46D
PARTNER_REL:         46B
PARTNER_TYPE:        3
KERNEL_REL:          46D
TRACE:
RFC_ROLE:            C
OWN_TYPE:            E
CPIC_CONVID:         31905351
```

In case you cannot run this sample program successfully, make sure that the system and user information has been changed to the correct values. Also check that the class path includes the JCo directory and the sample program itself. If that still does not help, there is some networking/configuration problem and you need to talk to your SAP Basis administrator.

4.2. Connection Pools

In (web) server applications, we usually use – at least to some extent – generic userids to log on to SAP. In that case it makes sense to use connection pooling. In this chapter, we will discuss how to use connection pools with JCo. For a general discussion on when and how to use connection pools, how to separate an application into generic and specific parts, etc., see the in-depth discussion on Connection Pooling in Appendix A-1.
A JCo connection pool is identified by its name and is global within the Java Virtual Machine (JVM). All connections in a pool use the same system/userid/client information. There can be as many pools as you need, though.
In sample program *TutorialConnect2* (remember: Appendix B contains the complete program listing), we will list the connection attributes the same way we did in *TutorialConnect1*. The only difference is that we now use connection pooling.

4.2.1. New classes introduced

JCO.Pool	Represents one connection pool.
JCO.PoolManager	Manages all connection pools within one JVM.

4.2.2. Selecting a pool name

```
static final String POOL_NAME = "Pool";
```

As far as JCo is concerned, you can use any pool name. Just remember that a pool is global within the JVM, so different applications running in the same JVM need to follow some naming standard to avoid unpleasant surprises.

4.2.3. Does the pool exist already?

```
JCO.Pool pool = JCO.getClientPoolManager().getPool(POOL_NAME);
  if (pool == null) {
```

All pools are managed by the global JCo *PoolManager* object (a singleton), which we can access via the *getClientPoolManager()* method of class *JCO*. The *getPool()* method tries to access a pool with the specific pool name. `null` is returned if no such pool exists. If we are sure that no pool with the given name already exists when we execute our code, then obviously we can skip the check and proceed to the pool creation immediately.

4.2.4. Creating a connection pool

```
OrderedProperties logonProperties =
  OrderedProperties.load("/logon.properties");
JCO.addClientPool(POOL_NAME,         // pool name
                  5,                 // maximum number of connections
                  logonProperties);  // properties
```

To create a new connection pool, we use method *addClientPool()*. The maximum number of connections specified can never be increased, so we have to choose a number large

enough for our application. Several overloaded versions of *addClientPool()* allow us to specify logon information in different ways. In this case, we have chosen to use a *Properties* object, created from a file using a utility class called *OrderedProperties* (a subclass of *Properties*, see Appendix B for the complete source code). Any other way of creating a *Properties* object could have been used instead.
The following is a sample `logon.properties` file:

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

Whenever we need an actual connection from the pool, we will borrow (acquire) it, make one or more calls to SAP, and finally return (release) the connection to the pool. For a detailed discussion of application design issues related to connection pools, refer to the Connection Pooling discussion in Appendix A-1.

```
mConnection = JCO.getClient(POOL_NAME);
```

The *getClient()* method is used to acquire a connection. JCo will either give us an existing open connection or open a new one for us – until we reach the limit of connections specified for the pool.
There is an overloaded version of *getClient()* with an additional parameter that is only required for R/3 3.1 systems. See the discussion of Using Connection Pools with R/3 3.1 in Appendix A-2.
If all connections in the pool are in use and the pool has reached its maximum size, JCo will wait for a certain time. If no connection has become available in the meantime, JCo will throw an exception with the group set to `JCO.Exception.JCO_ERROR_RESOURCE`. (See discussion of exception handling below). The default wait period is 30 seconds. You can change the value by calling the *setMaxWaitTime()* method available for the *PoolManager* as well as for individual *JCO.Pool* objects. The new value is passed in milliseconds.

```
  System.out.println(mConnection.getAttributes());
}
catch (Exception ex) {
  ex.printStackTrace();
}
finally {
  JCO.releaseClient(mConnection);
}
```

After our *getAttributes()* call is complete, we release the connection with *releaseClient()*. We normally put this into a `finally` block so that it will be executed regardless of whether or not an exception was thrown. Not releasing connections would eventually lead to big problems since all connections could become unavailable and no new requests requiring a connection could be processed any more.

In terms of our session with SAP, the session begins when we call *getClient()* and it ends when we call *releaseClient()*.
As long as our application is stateless – as far as SAP is concerned – we will always release the connection back to the pool as soon as we have finished with the SAP calls connected to one activity. Not necessarily after each SAP call, though. In other words, if you need to call multiple RFMs in a sequence, uninterrupted by user or other external interaction, you should keep the connection[11], and return it after all your calls are done. For a discussion of SAP state and commit handling, see Appendix A-5 on BAPIs, State, and Commit.
Note that when you use a connection pool, you never call the *connect()* or *disconnect()* methods of *JCO.Client*. The *PoolManager* takes care of all this as appropriate. If you are interested in knowing how long the *PoolManager* keeps connections to SAP open and how you can control that behavior, read Appendix A-3 about Dynamic Pool Management. Appendix A-4 discusses Pool Monitoring.

[11] Note that the SAP Basis System (now known as the SAP Web Application Server) automatically issues a DB commit statement at the end of each dialog step. Each RFM call is a separate dialog step.

5. The JCo Repository

The metadata of all RFMs we want to use must be available to JCo. This is accomplished by creating a *JCO.Repository* object. The actual metadata for the RFMs could be hard-coded into the Repository object (usually a bad idea) or retrieved dynamically from SAP at runtime (much better).
In sample program *TutorialBapi1*, we create a JCo Repository connected to SAP and execute two different BAPIs. We will have to learn how to deal with the different parameter types.

5.1. New Classes and Interfaces Introduced

JCO.Repository	Contains the runtime metadata for the RFMs.
IFunctionTemplate	Contains the metadata for one RFM.
JCO.Function	Represents an RFM with all its parameters.
JCO.ParameterList	Contains the import or export or table parameters of a JCO.Function.
JCO.Structure	Contains a structure.
JCO.Table	Contains a table.

5.2. Creating a JCo Repository

```
JCO.Repository mRepository;
  mRepository = new JCO.Repository("ARAsoft", mConnection);
```

We invoke the constructor for *JCO.Repository* with two parameters; the first one is an arbitrary name, the second one either a connection pool name or a *JCO.Client* object. In other words: Both connection pooling and direct connections are supported. In (web) server applications, we should always use a connection pool for the repository.
The userid used for the repository has to have sufficient authorizations in SAP for the metadata access to be possible. Please read the Javadoc for class *JCO.Repository* to find out which specific authorizations are required. You can use one special userid for the repository access and one or more "normal" userids for the actual application.

5.3. Creating a Function Object

```
public JCO.Function createFunction(String name) throws Exception {
  try {
    IFunctionTemplate ft =
      mRepository.getFunctionTemplate(name.toUpperCase());
    if (ft == null)
      return null;
    return ft.getFunction();
  }
```

```
  catch (Exception ex) {
    throw new Exception("Problem retrieving JCO.Function object.");
  }
}
```

Creating a *JCO.Function* object is a two-step process. First, we need to create a function template (interface *IFunctionTemplate*). A function template contains all the metadata (parameters and exceptions) for an RFM. JCo retrieves the metadata only once and caches it to optimize performance. The *getFunctionTemplate()* method of *JCO.Repository* is used to create the template. If `null` is returned, the RFM could not be found in SAP.
From the template, we can now create a *JCO.Function* object (method *getFunction()*). A function object not only has metadata, but also the actual parameters for the execution of the RFM. The relationship between a function template and a function in JCo is similar to the one between a class and an object in Java. The code shown above encapsulates the creation of a function object. It is a good idea to create a fresh function object for each individual execution. This way, you are sure that the parameters do not contain any leftovers from the previous call, like table rows that we should not really send (back) to SAP.

5.4. Executing a Function

In our sample program, we want to call *CompanyCode.GetList*[12] first. The underlying RFM is called `BAPI_COMPANYCODE_GETLIST`. There are no import parameters. We create the function object (using the utility method just introduced) and invoke the client object's *execute()* method, passing the function object as the parameter:

```
JCO.Function function = null;
function = this.createFunction("BAPI_COMPANYCODE_GETLIST");
mConnection.execute(function); }
```

This BAPI has two export parameters. First, we want to check the return message, which is contained in the `RETURN` structure export parameter. All parameters of a *JCO.Function* object can be accessed through the *getImportParameterList()*, *getExportParameterList()*, and *getTableParameterList()* methods. These methods return `null` if no parameters of the specific category exist for this function.
Within a parameter list, you access the individual parameters by their type and name. The *getStructure()* method allows us to access any structure parameter in an import or export parameter list.

[12] A fairly boring BAPI, but easy to use.

```
JCO.Structure returnStructure =
  function.getExportParameterList().getStructure("RETURN");
if (! (returnStructure.getString("TYPE").equals("") ||
       returnStructure.getString("TYPE").equals("S")) ) {
  System.out.println(returnStructure.getString("MESSAGE"));
  System.exit(1);
}
```

A structure consists of fields. Each field has a data type. ABAP uses different data types than Java, so some mapping must take place. The following table shows the different data types and the mapping between them.

ABAP Type	Description	Java Data Type	JCo Type Code
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

The *JCO.Structure* class contains type-specific getter methods, like *getString()* for strings. Normally an application will use the appropriate getter method, but JCo will try to convert the contents of the field to the data type you want, if you use a different getter method. Obviously, this conversion can fail (for example, when a string field contains "Willem Breuker" and you invoke *getDate()*). In that case, an exception will be thrown (see discussion below).
The following table lists all type-specific getter methods. In addition, *getValue()* can be used to access a field's contents generically. This method returns a Java *Object*.

JCo Type Code	JCo Access Method
JCO.TYPE_INT1	int getInt()
JCO.TYPE_INT2	int getInt()
JCO.TYPE_INT	int getInt()
JCO.TYPE_CHAR	String getString()
JCO.TYPE_NUM	String getString()
JCO.TYPE_BCD	BigDecimal getBigDecimal()
JCO.TYPE_DATE	Date getDate()
JCO.TYPE_TIME	Date getTime()
JCO.TYPE_FLOAT	double getDouble()
JCO.TYPE_BYTE	byte[ ] getByteArray()
JCO.TYPE_STRING	String getString()
JCO.TYPE_XSTRING	byte[ ] getByteArray()

In most cases, dealing with these data types is not a big problem.[13] You need to be aware of some peculiarities of the ABAP data types for date and time, though.
SAP has two distinct data types to deal with date/time information:

- ABAP data type T is a 6-byte string with format HHMMSS.
- ABAP data type D is an 8-byte string with format YYYYMMDD.

Both data types are used in RFMs, including the BAPIs. If a BAPI deals with a timestamp, two fields, one of type D and one of type T, will be used.
Java, on the other hand, uses one class, *Date* to represent both date and time information. Thus a timestamp can be represented in one variable.
JCo automatically converts between the ABAP and Java data types. Fields of ABAP data types D and T are represented as Java *Date* objects, leaving the unused portion of the *Date* object at its default value. It is up to the Java developer to know (looking it up in SAP at design time or using a metadata server at runtime) whether a given field holds an SAP date or time value and act accordingly.
The ABAP types D and T are somewhat lenient about what kind of values they allow. While there are no dates 00000000 or 99999999 in the real world, ABAP accepts these values for type D fields. And some BAPIs return these values. Making them illegal in JCo in order to conform to Java's more stringent requirements for *Date* objects would have prevented developers from using BAPIs that use strange dates. Here are the details of how JCo deals with these special values. You can assign "00000000" and "99999999" (or "0000-00-00" and "9999-99-99", if you prefer the ISO format) to a *JCO.Field* object using the *setString()* method. The *getString()* method will return "0000-00-00" or "9999-99-99", respectively. The *getDate()* method will return **null** for "00000000" (there is no suitable date that Java would support) and 9999-12-31 for "99999999".
A similar issue exists for time fields, "240000" is legal in ABAP, but not in Java.
JCo deals with this in the following manner:

[13] But remember the discussion above about BAPIs that do not follow the rules for currency amounts.

- The *setValue()* method now allows the strings "240000" as well as "24:00:00" for fields of ABAP type T.
- The *getString()* method for type T fields will return this value as "24:00:00".
- The *getDate()* method will return a *Date* object with the time set to 23:59:59 instead, since Java will not accept 24:00:00.

5.5. Accessing a Table

If the BAPI call was successful, we now want to output the table of all company codes. First, we retrieve the table, accessing the table parameter list (*getTableParameterList()*), and within the list, the concrete table (*getTable()*). The *JCO.Table* class has all the methods available for *JCO.Structure* and additional ones for navigation in the table. A table can have many rows but also no row at all.

```
JCO.Table codes = null;
codes =
  function.getTableParameterList().getTable("COMPANYCODE_LIST");
for (int i = 0; i < codes.getNumRows(); i++) {
  codes.setRow(i);
  System.out.println(codes.getString("COMP_CODE") + '\t' +
                     codes.getString("COMP_NAME"));
}
```

In the code above, we navigate by using the *setRow()* method to change the current row pointer to each row in the table in succession. Method *getNumRows()* tells us how many rows there are in total. Alternatively, we can use the *nextRow()* method. (The *JCO.Table* class also has the methods *previousRow()*, *firstRow()*, and *lastRow()*.) Here is the code rewritten to use *nextRow()* instead of *setRow()*:

```
JCO.Table codes = null;
codes =
  function.getTableParameterList().getTable("COMPANYCODE_LIST");
for (int i = 0; i < codes.getNumRows(); i++, codes.nextRow()) {
  System.out.println(codes.getString("COMP_CODE") + '\t' +
                     codes.getString("COMP_NAME"));
}
```

Methods *nextRow()* and *previousRow()* return a boolean value indicating whether there was a next or previous row to navigate to. Otherwise, the row pointer remains unchanged. Note that in a *JCO.Table* the row pointer (accessible via *getRow()*) always points to an actual row – as long as there is at least one row. There is no *before-the-first-row* or *after-the-last-row* status.
We access the fields within a table row in exactly the same way that we used to access fields within a structure. A structure is basically a special case of a table that has exactly one row.

5.6. Setting a Scalar Import Parameter

Next we want to call the *CompanyCode.GetDetail* BAPI for each company code (RFM `BAPI_COMPANYCODE_GETDETAIL`). This requires setting the scalar import `COMPANYCODEID` parameter. To access the import parameter list, we use *getImportParameterList()*. The value of the scalar parameter is set using the *setValue()* method, passing the value as the first, and the name as the second argument. Some people find this parameter sequence counter-intuitive, but that is how it works in JCo! Many overloaded version of *setValue()* exist in JCo, in order to support all the data types discussed above. Again, JCo will do its best to convert any value you pass to the data type appropriate for the field, and an exception is thrown if the conversion fails.
The *setValue()* method is also available for *JCO.Structure* and *JCO.Table* (and *JCO.Field*, see below) so that you can set the values of structure fields and fields in a table row.

```
codes.firstRow();
for (int i = 0; i < codes.getNumRows(); i++, codes.nextRow()) {
  function = this.createFunction("BAPI_COMPANYCODE_GETDETAIL");
  function.getImportParameterList().
    setValue(codes.getString("COMP_CODE"), "COMPANYCODEID");
  mConnection.execute(function);
  JCO.Structure returnStructure =
    function.getExportParameterList().getStructure("RETURN");
  if (! (returnStructure.getString("TYPE").equals("") ||
         returnStructure.getString("TYPE").equals("S") ||
         returnStructure.getString("TYPE").equals("W")) ) {
    System.out.println(returnStructure.getString("MESSAGE"));
  }
  JCO.Structure detail =
    function.getExportParameterList().
    getStructure("COMPANYCODE_DETAIL");
  System.out.println(detail.getString("COMP_CODE") + '\t' +
                     detail.getString("COUNTRY") + '\t' +
                     detail.getString("CITY"));
}
```

We invoke *firstRow()* before the loop since after the previous loop the row pointer for the table was at the last row. Note that in our BAPI error handling here, we accept not only empty string or "S" (success), but also "W" (warning) in our error handling. This is done since this particular BAPI sometimes issues a warning that address data (in structure parameter `COMPANYCODE_ADDRESS`) has not been maintained. We are not interested in this parameter in this sample program, anyway, therefore we can ignore this warning. In a production program, your error handling must be more elaborate and check for the

particular warning number. For a sample program, the code is good enough. Otherwise, the code should not contain anything unfamiliar.

6. Table Manipulation

We can access table fields, we can navigate through a table, but obviously in many applications we have to add rows or sometimes delete them. JCo has methods that allow us to do all this. Sample program *TutorialBapi2* demonstrates some of the features. Normally, we add table rows for table parameters that are sent to SAP (like adding line items for a sales order), but a complete example for that would take too much code. So instead, we play around with the table returned from *CompanyCode.GetList*.

```
codes.setRow(2);
codes.deleteRow();
codes.deleteRow(5);
codes.appendRow();
codes.setValue("XXXX", "COMP_CODE");
codes.setValue("Does not exist", "COMP_NAME");
codes.appendRows(2);
codes.setValue("YYYY", "COMP_CODE");
codes.setValue("Does not exist either", "COMP_NAME");
codes.nextRow();
codes.setValue("ZZZZ", "COMP_CODE");
codes.setValue("Nor does this", "COMP_NAME");
```

Method *deleteRow()* called with no parameter deletes the current row; when you specify a row number, the appropriate row will be deleted. Method *appendRow()* adds a row at the end of the table. You can pass an integer argument when you want multiple rows appended at the same time. This yields better performance than adding rows individually. Method *insertRow(int)* (not used in our program) allows you to insert a row anywhere in the table. Method *deleteAllRows()* deletes all rows of a table.
The following table summarizes the *JCO.Table* methods (that are not also available for a *JCO.Structure*) we have discussed:

JCO.Table Method	Description
int getNumRows()	Returns the number of rows.
void setRow(int pos)	Sets the current row pointer.
int getRow()	Returns the current row pointer.
void firstRow()	Moves to the first row.
void lastRow()	Moves to the last row.
boolean nextRow()	Moves to the next row.
boolean previousRow()	Moves to the previous row.
void appendRow()	Adds one row at the end of the table.
void appendRow(int num_rows)	Adds multiple rows at the end of the table.

void deleteAllRows()	Deletes all table rows.
void deleteRow()	Deletes the current row.
void deleteRow(int pos)	Deletes the specified row.
void insertRow(int pos)	Inserts a row at the specified position.

When you run the sample program, you will see error messages for the last three company codes since they – probably – do not exist in your SAP system.

7. Using Class JCO.Field

Structures contain fields, table rows contain fields, scalar parameters are fields. We have seen above how the appropriate classes all support methods for accessing and changing the contents of a field. Since there is all this commonality between the fields in the various contexts, JCo offers the class *JCO.Field* as a generic way to deal with a field. The classes *JCO.Structure*, *JCO.Table*, and *JCO.ParameterList* all have a *getField()* method to get hold of a field. Class *JCO.Field* itself offers all the getter and setter methods that we have discussed above. This abstraction can be pretty useful, when you want to build generic methods to deal with fields, regardless of where they originate.
A *JCO.Field* has metadata, including its name (method *getName()*), description (method *getDescription()*), data type (method *getType()*), length (method *getLength()*), and. number of decimals (method *getDecimals()*).
In addition, a field can have extended metadata (accessed though method *getExtendedFieldMetaData()*) that allows for advanced features when programming real applications.

8. Making Parameters Inactive

We have pretty much covered the basics of parameter handling. You know how to access structure, table and scalar parameters. You have heard of class *JCO.Field*. There is one additional tip that I want to give you, though, in order to allow you to further optimize performance (JCo is pretty fast anyway).
Many BAPIs have a large number of parameters, not all of which are always needed in a concrete application. There is no way to prevent a parameter from being returned from SAP, but you can avoid having it passed through to the Java layer by JCo. Simply declare the parameter as inactive, as shown below. This is particularly effective for large tables returned from SAP (in our example it is only a small structure, but every little bit helps).

```
function.getExportParameterList().
  setActive(false, "COMPANYCODE_ADDRESS");
```

9. Exception Handling

9.1. New Classes Introduced

JCO.Exception	The basic exception class.
JCO.ConversionException	Subclass for conversion errors.
JCO.AbapException	Subclass for exceptions thrown in the RFM.

9.2. How to Handle Exceptions

The most important thing first:
JCo throws exceptions that subclass *RuntimeException*. Just in case you forgot the implications, let me remind you that runtime exceptions do not have to be caught or specified in the method signature. This may be convenient, but it is also dangerous. I personally believe in local error handling, because then my program still knows the context in which the error occurred. So I recommend that you always use `try`/`catch` in your code even if the compiler does not complain if you don't.
JCo has only three exception types. *JCO.Exception* has two subclasses that you can catch separately if you want to. *JCO.Exception* has a *getGroup()* method that allows you to differentiate between different types of errors. See the Javadoc for the class for a list of all groups.
JCO.ConversionException is thrown whenever you call a getter or setter method that requires conversion and that conversion fails (Using *getDate()* on the name of a wonderful composer, for example, see above.)
JCO.AbapException occurs when the ABAP code invoked by you throws an exception. The BAPIs are not supposed to do this, but not all BAPIs follow the rules. The rule does not apply to non-BAPI RFMs, though, so we have to know how to deal with this contingency anyway. Let us assume that we just used the very useful RFM `DDIF_FIELDINFO_GET` in order to retrieve some structure or table metadata. This RFM will throw exception `NOT_FOUND` if we have passed an invalid name. The following code shows you how you could differentiate between different types of errors:

```
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) {
```

```
  ex.printStackTrace();
  System.exit(1);
}
catch (Exception ex) {
  ex.printStackTrace();
  System.exit(1);
}
```

There are three `catch` clauses:

- In the first one, we use method *getKey()* in order to access the exception string returned from SAP. If it is `NOT_FOUND`, we print a custom text; for any other exception, we use *getMessage()* to generate an appropriate text. This allows us to differentiate between ABAP exceptions that we want to handle in a specific way, and all the other ones that we want to handle generically. Remember that all possible exception strings are defined in SAP, so you can know them beforehand.
- The second `catch` deals with any other JCo-related problem. This would cover conversion exceptions (since we have no separate `catch` for them) and any other exception that occurred in JCo.
- The third `catch` handles any other exception that might have happened in our code.

This is just an example to introduce you to the possibilities. Depending on the requirements in your own code, you will have to adjust the exception handling accordingly.

10. Synchronization

In order to optimize performance, JCo itself synchronizes access only to *JCO.Pool* and *JCO.Repository* objects. Everything else is not synchronized. In a multi-threaded environment, you have to be careful when sharing objects (like *JCO.Table* objects) between different threads. *JCO.Client* objects acquired from a pool in one thread should never be used in a different thread. Also note that multiple concurrent SAP invocations for the same direct connection are not possible.

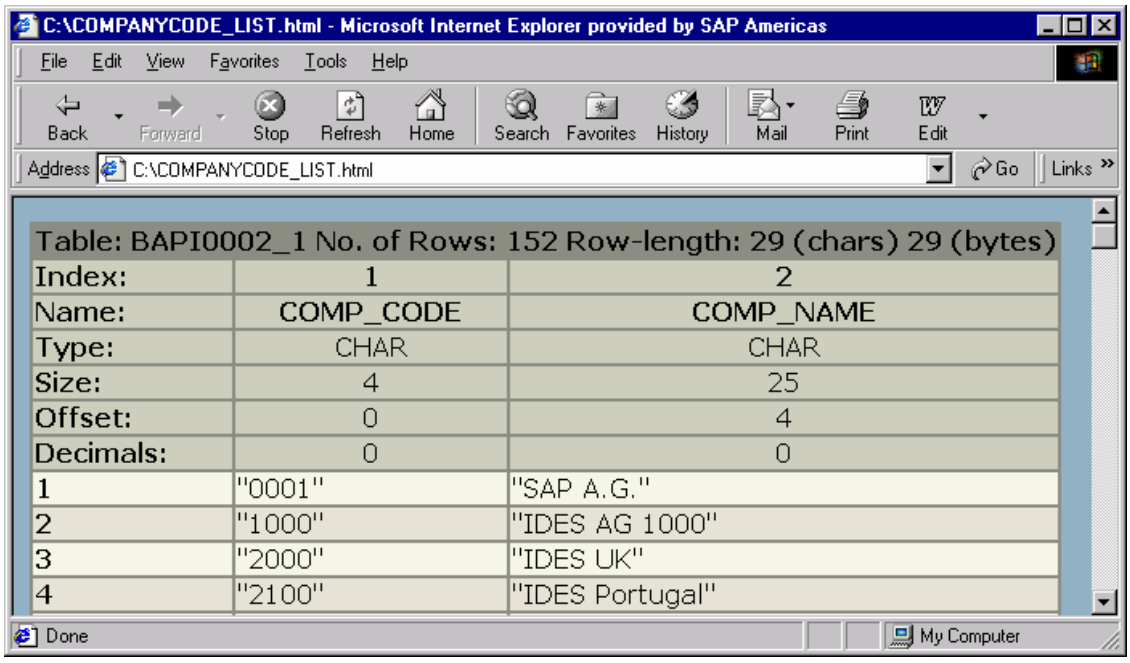
11. Debugging

When debugging an application, it is often advantageous to be able to quickly check the parameters being passed to and from SAP. JCo offers method *writeHTML()* to create an HTML file based on an object of type *JCO.Function*, *JCO.ParameterList*, *JCO.Structure*, or *JCO.Table*. For a table, by default, only the first 100 rows (and the last row) of a table are included, which saves space (and prevents the browser from aborting due to a very large HTML file). If you want more rows to be written out, you can set the global property, `jco.html.table_max_rows` to control the maximum number of rows put into the HTML file. Here is some sample code:

```
JCO.Function function =
  mRepository.getFunctionTemplate("BAPI_COMPANYCODE_GETLIST").
              getFunction();
mConnection.execute(function);
JCO.Table codes =
  function.getTableParameterList().getTable("COMPANYCODE_LIST");
String oldMaxRows = JCO.getProperty("jco.html.table_max_rows");
JCO.setProperty("jco.html.table_max_rows", "99999");
codes.writeHTML("c:\\COMPANYCODE_LIST.html");
JCO.setProperty("jco.html.table_max_rows", oldMaxRows);
```

Our code is polite enough to store the old value for the property before changing it. Then, after the call *to writeHTML()*, we restore the old value so that other users are not affected more than necessary (remember: the property is global within the Java Virtual Machine!). **Figure 1** shows a screen shot of the beginning of the generated HTML page.

Figure 1: A Table Parameter Displayed in an HTML Page



12. Further Information

For more details, study the JCo Javadocs, sign up for SAP's BIT526/CA926 training class, or ask us about custom JCo workshops. If you want to know something about ARAsoft's exciting extension to JCo (the **ARAsoft JCo Extension Library**), read Appendix A-6, and if you want a trial copy of the software discussed there, send me an email.

13. Contact Information

You can contact the author via

`tgs@arasoft.de`
`thomas.schuessler@sap.com`

A Appendix A: In-depth Discussions

A-1 Connection Pooling

Pools are critical for the performance of web server applications, but they can also be used in desktop applications. There are two benefits associated with using connection pools:

- You avoid the overhead of logging on to SAP, because once the logon has happened , the connection stays open and can be reused.
- You limit the maximum number of connections used concurrently, thus preventing the use of too many SAP resources. Be careful, though, not to create a performance bottleneck in your own application, by making the maximum number of connections too small and thus creating wait situations for your users. Your SAP system must be configured so that it can accommodate the extra load created by your application. This means that the number of SAP application servers and the configuration of your SAP gateway services have to be adjusted accordingly.

The fact that, within a pool, all connections use the same userid in the same SAP client in the same SAP system, has serious implications. To receive the maximum benefit from pooling, we would have to use only one userid for our complete application. Since SAP authorizations are linked to the userids, this is only applicable in some scenarios (usually applications where no update takes place in SAP and no security-relevant data is accessed).

Let us look first at a scenario where it does work: We want to build an Internet application that anybody on the Internet should be able to use. Users do not explicitly log on, they may not even know that there is an SAP system behind this application. Our application will connect to SAP on behalf of our users, and one generic userid will suffice since all our users will be given the same capabilities. In this scenario, pooling works beautifully. We limit the number of concurrent connections with SAP, thus also limiting the maximum performance impact our application can have on SAP.

A counter-example would be an employee self-service application where every employee has a separate SAP userid and has to identify himself in our application. In this kind of scenario, we would use a pool only for the *JCO.Repository* and other not security-critical read-only data access, but would employ direct connections for the specific application-related access of SAP where it is important to use the correct userid.

In many scenarios, we can combine the two approaches. Use a generic userid for basic read-only operations which require no special authorization level (like browsing our products in an Internet sales application). Since we know that there will be multiple concurrent users utilizing this part of our application, we will use a connection pool. For the more critical parts of our application, we will use specific userids (after asking the user to enter userid and password). Most likely there will only be one concurrent session per userid, so connection pooling does not make a lot of sense here.

A-2 Using Connection Pools with R/3 3.1

When we reuse an open connection in a pool, we do not know (at least in a web server scenario) who used the connection before. Some SAP functions maintain state. In order to isolate applications from one another, JCo calls the reset function in R/3 starting with release 4.0. This reset is performed automatically by JCo whenever a connection is returned to the pool. But R/3 3.1 does not have the reset function.
So when we run against a 3.1 system, we either have to reuse the existing connection with its state (if any) or disconnect and reconnect again, which is more costly in terms of performance. JCo offers a variant of the *getClient()* method with a boolean parameter that allows you to specify whether you want reset (disconnect/reconnect for a 3.1 system) or not. The *getClient()* method that does not have any parameters will **not** disconnect/reconnect. This should only be used if you are absolutely certain that no application using this pool ever calls any function in SAP that maintains state. Otherwise always pass `true` to the *getClient()* method!

A-3 Dynamic Pool Management

The size of a pool should normally be large enough to accommodate periods of peek activity in our application. Does that mean that a connection, once opened, stays open forever? No, JCo will close connections that have not been used in a while. This happens only to connections that no-one has acquired via *getClient()*, though! You do not have to worry that JCo closes a connection before you have released it back to the pool. JCo will close an idle, non-acquired connection in a pool after the connection timeout period (*getConnectionTimeout()*) has expired. The default value for this is 600,000 ms (10 minutes). You can use method *setConnectionTimeout()* to assign a new value.
How often does JCo check pooled connection for the connection timeout period? You can control that, too. The default is that JCo checks its pooled connections every 60,000 ms (1 minute). Use *getTimeoutCheckPeriod()* to retrieve the value and *setTimeoutCheckPeriod()* to change it. Normally, the defaults for both these values should suffice, though.

A-4 Pool Monitoring

In order to monitor the activities in a pool, an application can register a *JCO.PoolChangedListener* using the *addPoolChangedListener()* method of *JCO.PoolManager*. The listener can use the following methods available for a *JCO.Pool* object to find out information useful for tracing and tuning purposes:

JCO.Pool Monitoring Methods	Description
int getMaxPoolSize()	Returns the maximum size of the pool (as specified at pool creation time).
int getMaxUsed()	Returns the maximum number of open connections ever reached.
int getCurrentPoolSize()	Returns the current number of open connections.
int getNumUsed()	Returns the current number of connections acquired by the application clients (via getClient()).
byte[ ] getStates()	Returns a byte array with the state of each object in the connection pool described in one byte. See the Javadoc for class JCO for a list of the individual state bits.

A-5 BAPIs, State, and Commit

Most RFMs, including most BAPIs, are stateless. SAP does not remember anything between calls in the same session (connection). As stated before, with a connection pool, we should release the connection back to the pool as soon as we are finished with one uninterrupted activity in our application – if all the RFMs we invoke are stateless.
On the other hand, most updating BAPIs require an additional external commit call to actually cause any change on the SAP database to happen. This allows us to combine multiple update BAPI calls[14] into one Logical Unit of Work (LUW). The commit call must happen in the same SAP session in which we called the update BAPI(s) (via a call to *BapiService.TransactionCommit*, RFM name `BAPI_TRANSACTION_COMMIT`). In other words: whenever you deal with stateful BAPIs you need to hold on to your connection until you have reached the end of your LUW.

A-6 The ARAsoft JCo Extension Library

JCo is a wonderful product and allows you to build applications easily. But there are some issues with BAPI programming that JCo does not know about.

- BAPIs use lots of codes. When we listed detail information for each company code in one of our example programs, for example, we got a country code, but not the name of the country. There are hundreds of cases like this. What if we need a list of all country codes and names for a GUI? SAP allows you to get the required information, but it is very cumbersome, release-dependent, and slow. There should be a component that takes care of this.

[14] This requires that all involved BAPIs follow the rules regarding commit handling specified in the SAP BAPI Programming Guide. Not all BAPIs do this. Please check the documentation of each BAPI.

- BAPIs require the internal format of those fields that have associated conversion routines in SAPGUI. The client program must deal with the necessary conversions, since most BAPIs will not. Again, this can be done, but an application program should not have to know these things.
- Different BAPIs use different structures for return messages. Field names have changed. A generic way of dealing with return messages is required.
- Not all BAPIs follow the rules regarding currency amounts. As a result, sometimes you may get invalid amounts without noticing it. There needs to be a generic solution to this.
- There is lots of useful metadata (like texts to build multilingual GUIs) in SAP. It should be easy to access it, and the metadata should be cached to avoid performance problems in SAP.
- Fields and error messages have extensive documentation in SAP. An application should be able to access the documentation without much effort.

Solving all these issues and some more is the purpose of our library. Let me show you some sample code that demonstrates some of the features.
Here is an example of how we access the name of a country easily.

```
JCO.Field countryCode = detail.getField("COUNTRY");
String countryName = mRepository.getDescriptionForValue(countryCode);
```

Here we see generalized error message handling and documentation access.

```
BapiMessageInfo bapiMessage = new BapiMessageInfo(returnStructure);
// Warning FN021 can be ignored in our case
  if ( ! bapiMessage.
         isBapiReturnCodeOkay(false, false, null, "FN021") ) {
    System.out.println(bapiMessage.getFormattedMessage());
    System.out.println("--- Documentation for error message: ---");
// One line of code retrieves the documentation.
  String[] documentation =
    mRepository.getMessageDocumentation(bapiMessage);
  for (int j = 0; j < documentation.length; j++) {
    System.out.println(documentation[j]);
  }
}
```

You want the documentation for a field in SAP?

```
JCO.Function function =
  mRepository.createFunction("BAPI_CUSTOMER_GETDETAIL2");
JCO.Field custField =
  function.getImportParameterList().getField("CUSTOMERNO");
// One line of code retrieves the documentation.
String[] documentation = mRepository.getFieldDocumentation(custField);
```

You want to automatically convert a value so that the BAPI will understand it?

```
  JCO.Function function =
    mRepository.createFunction("BAPI_CUSTOMER_GETDETAIL2");
  JCO.Field custField =
    function.getImportParameterList().getField("CUSTOMERNO");
  custField.setValue(customerCode);        // entered by the user
// This BAPI needs the internal format, so we convert.
// But we do not have to know whether conversion
// would really be necessary, the method will always work.
  String custCode = mRepository.getInternalValue(custField);
  custField.setValue(custCode);
```

You want a collection of all Helpvalues for a field?

```
JCO.Function function =
   mRepository.createFunction("BAPI_CUSTOMER_GETDETAIL2");
JCO.Structure custDetail =
  function.getExportParameterList().getStructure("CUSTOMERADDRESS");
JCO.Field countryField = custDetail.getField("COUNTRY");
GenericCollection helpValues = mRepository.getHelpvalues(countryField);
```

You want the correct amount from a BAPI that breaks the rules?

```
IMetaData md = mRepository.getStructureDefinition("BAPISDSTAT");
JCO.Structure s = new JCO.Structure(md);
JCO.Field f = s.getField("NET_VALUE");
f.setValue("1234.56");   // simulate some data
BigDecimal bd = mRepository.getCorrectAmount(f, "JPY");
```

How is all this possible? ARAsoft has spent years developing the ARAsoft Java BAPI ObjectFactory that encapsulates all the generic functionality required for BAPI programming that we could think of. The big step forward then was to make all this available even more easily by creating a subclass of *JCO.Repository* so that Java programmers could access the functionality in very little code and with very little extra training. Interested? Please contact us…

A-7 The ARAsoft JCo Proxy Generator

This is part of the latest version of the ARAsoft JCo Extension Library. You can generate proxies for BAPIs and other RFMs. The BAPI proxies fully exploit object-orientation and the additional metadata available in the SAP Business Object Repository. All proxies allow you full access to the features of JCO.Table, JCO.Structure, and JCO.Field, so you still benefit from your JCo knowledge. On the other hand, you do not need to type function and field names anymore. The generated proxy classes are fully typed and therefore support Code Insight for BAPIs, key fields, parameters, and fields. Also, you do not need to worry about import, export, and table parameters anymore. The proxies just expose two types of parameters, request and response.
Optionally, you can have the Proxy Generator include Javadoc comments for

- BOR object type documentation
- BAPI documentation
- Field documentation

This can be useful if you want to review the documentation inside your Java IDE.
The Proxy Generator can optionally exclude BAPIs that are obsolete, not released, or that use SAPGUI dialogs. This makes it easier to avoid those BAPIs in your applications.
If you use JBuilder, you can use the Proxy Generator Wizard, which is fully integrated with JBuilder.
Currently, the minimum SAP release for which the Proxy Generator works, is 4.0A, but if there is enough interest, we will also support 3.1.
Let me show you some client code utilizing the proxies:

```
JCO.Client mConnection;
JCoRepository mRepository;
JCoContext context;
  mConnection.connect();
  mRepository = new JCoRepository(mConnection);
  context = new JCoContext(mConnection, mRepository);
  SAPCompanyCode companyCode = new SAPCompanyCode(context);
// We do not want to set request parameters
  SAPCompanyCode.GetListResponse responseGetList =
    companyCode.getList(null);
  SAPCompanyCode.GetListResponse.CompanyCodeList codes =
    responseGetList.getCompanyCodeList();
  for (int i = 0; i < codes.getNumRows(); i++, codes.nextRow()) {
    companyCode.getKeyFields().getCompanyCodeId().
                               setValue(codes.getCOMP_CODE());
    SAPCompanyCode.GetDetailResponse responseGetDetail =
      companyCode.getDetail();
    SAPCompanyCode.GetDetailResponse.CompanycodeDetail
```

```
      detail = responseGetDetail.getCompanycodeDetail();
    JCO.Field countryCode = detail.getJCoFields().getCOUNTRY();
    String countryName =
      mRepository.getDescriptionForValue(countryCode);
    System.out.println(detail.getCOMP_CODE() + '\t' +
                       countryCode.getString() + " (" +
                       countryName
                       + ")" + '\t' +
                       detail.getCITY());
  }
```

If you like this better than native code, please send me an email!

A-8 Creating a Sales Order

In this appendix I will walk you through a complete program that creates a sales order in SAP with as few fields populated as possible, discussing a few important issues along the way. The program should actually work for any IDES client, but I obviously cannot guarantee that.
We are using proxies here since the code should be easier to enter and also understand this way.

```
package de.arasoft.demo.jco;

import com.sap.mw.jco.*;
import de.arasoft.sap.jco.*;
import de.arasoft.sap.interfacing.*;
import sap.generated.*;

/**
 *
 * Sample program using the BAPI proxies
 * to create a sales order.
 *
 * http://www.arasoft.de
 * tgs@arasoft.de
 * Copyright (c) 2002 ARAsoft GmbH
 *
 * @author Thomas G. Schuessler, ARAsoft GmbH
 * @version 1.0
 */

public class DemoSalesOrder extends Object {

  JCO.Client mConnection;

// This is the ARAsoft extension of JCO.Repository
  JCoRepository mRepository;
```

Class *JCoContext* has two major functions:

- It hides the differences of using direct connections or pools from the rest of the application.
- It helps you to manage state in SAP (see below).

```
  JCoContext context;
```

```
  public DemoSalesOrder() {
    try {
// Change the logon information to your own system/user
      mConnection =
        JCO.createClient(
          "001",                // SAP client
          "<userid>",           // userid
          "*****",              // password
          "EN",                 // language
          "<hostname>",         // application server host name
          "00");                // system number

      mConnection.connect();
```

We are using our subclass of SAP's *JCO.Repository* (*JCoRepository*) here since it provides very useful methods that we will need later.

```
      mRepository = new JCoRepository(mConnection);

// JCoContext can be created via a direct connection (JCO.Client)
// or a pool.
      context = new JCoContext(mConnection, mRepository);
```

If we need to maintain state in SAP because we want to make multiple calls in the same session, the behavior for direct connections and pools must be very different. Class *JCoContext* hides all the differences, we just need to tell the object that from now on we want to be stateful.

```
// Start an SAP transaction so that we could, if we wanted, commit
// multiple updates
// in one SAP transaction. Also guarantees that if we use a connection
// pool, we call
// commit in the same SAP session (connection) in which we called
// the update BAPI.
      context.startSapTransaction();

// Instantiate the BAPI proxy class
      SAPSalesOrder salesOrder = new SAPSalesOrder(context);

// Create the request container
```

```
      SAPSalesOrder.CreateFromDat1Request request =
        salesOrder.getRequests().createCreateFromDat1Request();

// Most BAPIs doing DB updates do not ever commit themselves.
// This BAPI allows us to choose. In this sample, we want to
// control the transaction ourselves.
      request.getWithoutCommit().setValue("X");

// Set some sales order header fields
      SAPSalesOrder.CreateFromDat1Request.OrderHeaderIn orderHeader =
        request.getOrderHeaderIn();
      orderHeader.setSALES_ORG("3000");
      orderHeader.setDISTR_CHAN("10");
      orderHeader.setDIVISION("00");
```

We need an order type. In a GUI-driven program, we would first offer the user a list of available codes and associated descriptions. This is how we would get the data:

```
//      GenericCollection hv =
// mRepository.getHelpvalues(orderHeader.getJCoFields().getDOC_TYPE());
```

For now we hard-code an English code, and then convert to the internal format:

```
// The order type uses different codes in different languages.
// Assuming that the
// user enters the value (in English in this sample program),
// we need to convert.
      String userEnteredOrderType = "OR";

// Conversion: Alternative 1
      orderHeader.setDOC_TYPE(userEnteredOrderType);
      orderHeader.setDOC_TYPE(
        mRepository.getInternalValue(
        orderHeader.getJCoFields().getDOC_TYPE()));

// Conversion: Alternative 2
//      String internalOrderType =
// mRepository.getHelpvalues(orderHeader.getJCoFields().getDOC_TYPE()).
// getExternalItem(userEnteredOrderType).getInternalCode();
//      orderHeader.setDOC_TYPE(internalOrderType);
```

```
      orderHeader.setPURCH_NO("Purchase Order 1");


// Specify the sold-to customer
      SAPSalesOrder.CreateFromDat1Request.OrderPartners orderPartners =
        request.getOrderPartners();
      orderPartners.appendRow();
```

If the user needs choice here, we would do another Helpvalues call to retrieve a list of legal codes and descriptions. We have seen that and conversion before, so this time we hard-code the internal code.

```

// Could also be converted from the code in a different language, like
// we did for the order type.
      orderPartners.setPARTN_ROLE("AG");
```

But hard-coding customer numbers would be strange, so here we assume user input and convert it to the internal format.

```
// The customer number must be converted into the internal format
required by the BAPI.
      String userEnteredCustomerNumber = "30001";
      orderPartners.setPARTN_NUMB(userEnteredCustomerNumber);
      orderPartners.setPARTN_NUMB(
        mRepository.getInternalValue(
        orderPartners.getJCoFields().getPARTN_NUMB()));
```

Material numbers need to be converted, too:

```
// Add two materials; convert the material numbers.
      SAPSalesOrder.CreateFromDat1Request.OrderItemsIn orderItems =
        request.getOrderItemsIn();
      orderItems.appendRow();
      orderItems.setMATERIAL("M-08");
      orderItems.setMATERIAL(
        mRepository.getInternalValue(
        orderItems.getJCoFields().getMATERIAL()));
```

Really strange. The only way to know this is to read the field description in SAP.

```
// Idiosyncrasy of the BAPI: expects three implicit decimals
      orderItems.setREQ_QTY("1000");   // Really: 1.000
```

```
      orderItems.appendRow();
      orderItems.setMATERIAL("DCC-12");
      orderItems.setMATERIAL(
        mRepository.getInternalValue(
        orderItems.getJCoFields().getMATERIAL()));

// Idiosyncrasy of the BAPI: expects three implicit decimals
      orderItems.setREQ_QTY("15000");  // Really: 15.000

// Call the BAPI
      SAPSalesOrder.CreateFromDat1Response response =
        salesOrder.createFromDat1(request);

// Check the return code message
      BapiMessageInfo returnMessage =
        new BapiMessageInfo(response.getReturn().getJCoStructure());
      if ( ! returnMessage.isBapiReturnCodeOkay() ) {

// Call rollback just in case.
```

We failed, so we want to roll back and possibly start again with a clean slate.

```
        context.rollbackSapTransaction();

        System.out.println(returnMessage.getFormattedMessage());
        System.out.println("--- Documentation for error message: ---");
        String[] documentation =
          mRepository.getMessageDocumentation(returnMessage);
        for (int j = 0; j < documentation.length; j++) {
          System.out.println(documentation[j]);
        }
      } else {

// Call commit so the DB is updated.
```

The BAPI call succeeded. Let us commit the update!

```
        context.commitSapTransaction();
```

The sales document number of the newly created sales order is returned as the key field of the proxy object.

```
     System.out.println("Sales order number of new sales order is: " +
        System.out.println(
                    "Sales order number of new sales order is: "
                    + salesOrder.getKeyFields().
                        getSalesDocument().getString());
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    finally {
```

Remember: you could use a connection pool in this program without changing any of the actual application code.

```
     System.out.println("Sales order number of new sales order is: " +
      mConnection.disconnect();
    }
  }

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

B Appendix B: Code Listings

B-1 Sample Program TutorialConnect1

```
import com.sap.mw.jco.*;
/**
 * @author Thomas G. Schuessler, ARAsoft GmbH
 * http://www.arasoft.de
 */
public class TutorialConnect1 extends Object {
  JCO.Client mConnection;
  public TutorialConnect1() {
    try {
      // Change the logon information to your own system/user
      mConnection =
        JCO.createClient("001", // SAP client
          "<userid>",           // userid
          "*****",              // password
          null,                 // language
          "<hostname>",         // application server host name
          "00");                // system number
      mConnection.connect();
      System.out.println(mConnection.getAttributes());
      mConnection.disconnect();
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
  }
  public static void main (String args[]) {
    TutorialConnect1 app = new TutorialConnect1();
  }
}
```

B-2 Sample Program TutorialConnect2

```
import com.sap.mw.jco.*;
/**
 * @author Thomas G. Schuessler, ARAsoft GmbH
 * http://www.arasoft.de
 */
public class TutorialConnect2 extends Object {
  static final String POOL_NAME = "Pool";
  JCO.Client mConnection;
  public TutorialConnect2() {
    try {
      JCO.Pool pool = JCO.getClientPoolManager().getPool(POOL_NAME);
      if (pool == null) {
        OrderedProperties logonProperties =
          OrderedProperties.load("/logon.properties");
        JCO.addClientPool(POOL_NAME,  // pool name
                          5,          // maximum number of connections
                          logonProperties);     // properties
      }
      mConnection = JCO.getClient(POOL_NAME);
      System.out.println(mConnection.getAttributes());
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
    finally {
      JCO.releaseClient(mConnection);
    }
  }
  public static void main (String args[]) {
    TutorialConnect2 app = new TutorialConnect2();
  }
}
```

B-3 Utility 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 IOException {
    OrderedProperties props = null;
    java.io.InputStream is =
      OrderedProperties.class.getResourceAsStream(name);
    if ( is != null ) {
      props = new OrderedProperties();
      props.load(is);
      return props;
    } else {
      if ( ! name.startsWith("/") ) {
        return load("/" + name);
      } 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;
  }
}
```

B-4. Sample Program TutorialBapi1

```
import com.sap.mw.jco.*;
/**
 * @author Thomas G. Schuessler, ARAsoft GmbH
 * http://www.arasoft.de
 */
public class TutorialBapi1 extends Object {
  JCO.Client mConnection;
  JCO.Repository mRepository;
  public TutorialBapi1() {
    try {
      // Change the logon information to your own system/user
      mConnection =
        JCO.createClient("001", // SAP client
          "<userid>",           // userid
          "*****",              // password
          null,                 // language
          "<hostname>",         // application server host name
          "00");                // system number
      mConnection.connect();
      mRepository = new JCO.Repository("ARAsoft", mConnection);
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    JCO.Function function = null;
    JCO.Table codes = null;
    try {
      function = this.createFunction("BAPI_COMPANYCODE_GETLIST");
      if (function == null) {
        System.out.println("BAPI_COMPANYCODE_GETLIST" +
                           " not found in SAP.");
        System.exit(1);
      }
      mConnection.execute(function);
```

```
      JCO.Structure returnStructure =
        function.getExportParameterList().getStructure("RETURN");
      if (! (returnStructure.getString("TYPE").equals("") ||
             returnStructure.getString("TYPE").equals("S")) ) {
        System.out.println(returnStructure.getString("MESSAGE"));
        System.exit(1);
      }
      codes =
        function.getTableParameterList().getTable("COMPANYCODE_LIST");
      for (int i = 0; i < codes.getNumRows(); i++) {
        codes.setRow(i);
        System.out.println(codes.getString("COMP_CODE") + '\t' +
                           codes.getString("COMP_NAME"));
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    try {
      codes.firstRow();
      for (int i = 0; i < codes.getNumRows(); i++, codes.nextRow()) {
        function = this.createFunction("BAPI_COMPANYCODE_GETDETAIL");
        if (function == null) {
          System.out.println("BAPI_COMPANYCODE_GETDETAIL" +
                             " not found in SAP.");
          System.exit(1);
        }
        function.getImportParameterList().
          setValue(codes.getString("COMP_CODE"), "COMPANYCODEID");
        function.getExportParameterList().
          setActive(false, "COMPANYCODE_ADDRESS");
        mConnection.execute(function);
        JCO.Structure returnStructure =
          function.getExportParameterList().getStructure("RETURN");
        if (! (returnStructure.getString("TYPE").equals("") ||
               returnStructure.getString("TYPE").equals("S") ||
```

```
              returnStructure.getString("TYPE").equals("W")) ) {
          System.out.println(returnStructure.getString("MESSAGE"));
        }
        JCO.Structure detail =
          function.getExportParameterList().
          getStructure("COMPANYCODE_DETAIL");
        System.out.println(detail.getString("COMP_CODE") + '\t' +
                           detail.getString("COUNTRY") + '\t' +
                           detail.getString("CITY"));
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    mConnection.disconnect();
  }
  public JCO.Function createFunction(String name) throws Exception {
    try {
      IFunctionTemplate ft =
        mRepository.getFunctionTemplate(name.toUpperCase());
      if (ft == null)
        return null;
      return ft.getFunction();
    }
    catch (Exception ex) {
      throw new Exception("Problem retrieving JCO.Function object.");
    }
  }
  public static void main (String args[]) {
    TutorialBapi1 app = new TutorialBapi1();
  }
}
```

B-5. Sample Program TutorialBapi2

```
import com.sap.mw.jco.*;
/**
 * @author Thomas G. Schuessler, ARAsoft GmbH
 * http://www.arasoft.de
 */
public class TutorialBapi2 extends Object {
  JCO.Client mConnection;
  JCO.Repository mRepository;
  public TutorialBapi2() {
    try {
      // Change the logon information to your own system/user
      mConnection =
        JCO.createClient("001", // SAP client
          "<userid>",           // userid
          "*****",              // password
          null,                 // language
          "<hostname>",         // application server host name
          "00");                // system number
      mConnection.connect();
      mRepository = new JCO.Repository("ARAsoft", mConnection);
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    JCO.Function function = null;
    JCO.Table codes = null;
    try {
      function = this.createFunction("BAPI_COMPANYCODE_GETLIST");
      if (function == null) {
        System.out.println("BAPI_COMPANYCODE_GETLIST" +
                           " not found in SAP.");
        System.exit(1);
      }
      mConnection.execute(function);
```

```
      JCO.Structure returnStructure =
        function.getExportParameterList().getStructure("RETURN");
      if (! (returnStructure.getString("TYPE").equals("") ||
             returnStructure.getString("TYPE").equals("S")) ) {
        System.out.println(returnStructure.getString("MESSAGE"));
        System.exit(1);
      }
      codes =
        function.getTableParameterList().getTable("COMPANYCODE_LIST");
      codes.setRow(2);
      codes.deleteRow();
      codes.deleteRow(5);
      codes.appendRow();
      codes.setValue("XXXX", "COMP_CODE");
      codes.setValue("Does not exist", "COMP_NAME");
      codes.appendRows(2);
      codes.setValue("YYYY", "COMP_CODE");
      codes.setValue("Does not exist either", "COMP_NAME");
      codes.nextRow();
      codes.setValue("ZZZZ", "COMP_CODE");
      codes.setValue("Nor does this", "COMP_NAME");
      for (int i = 0; i < codes.getNumRows(); i++) {
        codes.setRow(i);
        System.out.println(codes.getString("COMP_CODE") + '\t' +
                           codes.getString("COMP_NAME"));
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    try {
      codes.firstRow();
      for (int i = 0; i < codes.getNumRows(); i++, codes.nextRow()) {
        function = this.createFunction("BAPI_COMPANYCODE_GETDETAIL");
        if (function == null) {
          System.out.println("BAPI_COMPANYCODE_GETDETAIL" +
```

```
                            " not found in SAP.");
        System.exit(1);
      }
      function.getImportParameterList().
        setValue(codes.getString("COMP_CODE"), "COMPANYCODEID");
      mConnection.execute(function);
      JCO.Structure returnStructure =
        function.getExportParameterList().getStructure("RETURN");
      if (! (returnStructure.getString("TYPE").equals("") ||
             returnStructure.getString("TYPE").equals("S") ||
             returnStructure.getString("TYPE").equals("W")) ) {
        System.out.println(returnStructure.getString("MESSAGE"));
      }
      JCO.Structure detail =
        function.getExportParameterList().
        getStructure("COMPANYCODE_DETAIL");
      System.out.println(detail.getString("COMP_CODE") + '\t' +
                         detail.getString("COUNTRY") + '\t' +
                         detail.getString("CITY"));
    }
  }
  catch (Exception ex) {
    ex.printStackTrace();
    System.exit(1);
  }
  mConnection.disconnect();
}
public JCO.Function createFunction(String name) throws Exception {
  try {
    IFunctionTemplate ft =
      mRepository.getFunctionTemplate(name.toUpperCase());
    if (ft == null)
      return null;
    return ft.getFunction();
  }
  catch (Exception ex) {
    throw new Exception("Problem retrieving JCO.Function object.");
```

```
    }
  }
  public static void main (String args[]) {
    TutorialBapi2 app = new TutorialBapi2();
  }
}
```

B-6. Sample Program TutorialBapi1a

```
import com.sap.mw.jco.*;
import de.arasoft.sap.jco.*;
import de.arasoft.sap.interfacing.*;
/**
 * Uses the ARAsoft JCo Extension Library which adds many
 * features useful for BAPI programming.
 * @author Thomas G. Schuessler, ARAsoft GmbH
 * http://www.arasoft.de
 */
public class TutorialBapi1a extends Object {
  JCO.Client mConnection;
// This is the ARAsoft extension of JCO.Repository
  JCoRepository mRepository;
  public TutorialBapi1a() {
    try {
      // Change the logon information to your own system/user
      mConnection =
        JCO.createClient("001", // SAP client
          "<userid>",           // userid
          "*****",              // password
          null,                 // language
          "<hostname>",         // application server host name
          "00");                // system number
      mConnection.connect();
      mRepository = new JCoRepository(mConnection);
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    JCO.Function function = null;
    JCO.Table codes = null;
    try {
      function =
        mRepository.createFunction("BAPI_COMPANYCODE_GETLIST");
```

```
      if (function == null) {
        System.out.println("BAPI_COMPANYCODE_GETLIST" +
                           " not found in SAP.");
        System.exit(1);
      }
// Using this, the code needs not to be changed for connection pools.
      mRepository.executeStateless(function);
// Check the BAPI return message
      JCO.Structure returnStructure =
        function.getExportParameterList().getStructure("RETURN");
// BapiMessageInfo hides all the differences between the various
// structures that SAP uses for the RETURN parameter of the BAPIs.
      BapiMessageInfo bapiMessage =
        new BapiMessageInfo(returnStructure);
      if ( ! bapiMessage.isBapiReturnCodeOkay() ) {
        System.out.println(bapiMessage.getFormattedMessage());
        System.out.println("--- Documentation for error message: ---");
// One line of code retrieves the documentation.
        String[] documentation =
          mRepository.getMessageDocumentation(bapiMessage);
        for (int i = 0; i < documentation.length; i++) {
          System.out.println(documentation[i]);
        }
        System.exit(1);
      }
      codes =
        function.getTableParameterList().getTable("COMPANYCODE_LIST");
      for (int i = 0; i < codes.getNumRows(); i++) {
        codes.setRow(i);
        System.out.println(codes.getString("COMP_CODE") + '\t' +
                           codes.getString("COMP_NAME"));
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
```

```
    try {
      codes.firstRow();
      for (int i = 0; i < codes.getNumRows(); i++, codes.nextRow()) {
        function =
          mRepository.createFunction("BAPI_COMPANYCODE_GETDETAIL");
        if (function == null) {
          System.out.println("BAPI_COMPANYCODE_GETDETAIL" +
                             " not found in SAP.");
          System.exit(1);
        }
        function.getImportParameterList().
          setValue(codes.getString("COMP_CODE"), "COMPANYCODEID");
        mConnection.execute(function);
// Check the BAPI return message
        JCO.Structure returnStructure =
          function.getExportParameterList().getStructure("RETURN");
        BapiMessageInfo bapiMessage =
          new BapiMessageInfo(returnStructure);
// Warning FN021 can be ignored in our case
        if ( ! bapiMessage.
               isBapiReturnCodeOkay(false, false, null, "FN021") ) {
          System.out.println(bapiMessage.getFormattedMessage());
          System.out.println
            ("--- Documentation for error message: ---");
// One line of code retrieves the documentation.
          String[] documentation =
            mRepository.getMessageDocumentation(bapiMessage);
          for (int j = 0; j < documentation.length; j++) {
            System.out.println(documentation[j]);
          }
        }
        JCO.Structure detail =
          function.getExportParameterList().
          getStructure("COMPANYCODE_DETAIL");
        JCO.Field countryCode = detail.getField("COUNTRY");
// One line of code to retrieve the description text
// (in this case: the country name)
```

```
        String countryName =
          mRepository.getDescriptionForValue(countryCode);
        System.out.println(detail.getString("COMP_CODE") + '\t' +
                           countryCode.getString() + " (" +
                           countryName
                           + ")" + '\t' +
                           detail.getString("CITY"));
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    mConnection.disconnect();
  }
  public static void main (String args[]) {
    TutorialBapi1a app = new TutorialBapi1a();
  }
}
```