



SAP Java Connector (Standalone Version)

Release 3.1

Copyright

© Copyright 2019 SAP SE. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE. The information contained herein may be changed without prior notice.

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, OS/2, Parallel Sysplex, MVS/ESA, AIX, S/390, AS/400, OS/390, OS/400, iSeries, pSeries, xSeries, zSeries, z/OS, AFP, Intelligent Miner, WebSphere, Netfinity, Tivoli, Informix, i5/OS, POWER, POWER5, OpenPower and PowerPC are trademarks or registered trademarks of IBM Corporation.

Adobe, the Adobe logo, Acrobat, PostScript, and Reader are either trademarks or registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Oracle Corporation.





JavaScript is a registered trademark of Oracle Corporation.

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE in Germany and in several other countries all over the world. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP SE and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Icons in Body Text

Icon	Meaning
	Caution
	Note
	Recommendation
	Example

Typographic Conventions

Type Style	Description
<i>Example text</i>	Emphasis
<code>Example text</code>	File, package and directory names and their paths, names of classes, variables and parameters, and names of installation, upgrade and database tools
Example text	Property values for variables, etc.

Document Version

Version 1.0	10-25-2019
--------------------	-------------------

TABLE OF CONTENTS

SAP JAVA CONNECTOR	5
SAP JCo Functions.....	5
SAP JCo Architecture.....	6
SAP JCo Installation	6
CLIENT PROGRAMMING	9
Establishing a Connection to an AS ABAP	9
Executing Function Modules in an SAP System	12
Using Function Module Parameters.....	14
Using Multi-Threading	17
SAP JCo Repository	18
Mapping of ABAP and Java Data Types	19
Using suitable getter methods	20
Generic processing of fields	21
Optimizing performance by deactivating parameters	21
JSON Import/Export of JCoRecord	22
Exception Handling	24
SERVER PROGRAMMING	26
Calling an RFC Server from AS ABAP	26
Java Program for Starting an RFC Server	27
Implementing an Exception Listener	29
Monitoring Server Connections	30
Processing an ABAP Call	31
Stateful Server Calls	32
IDOC SUPPORT FOR EXTERNAL JAVA APPLICATIONS	34
Features	34
Implementation Considerations	34
Further Information.....	34

SAP JAVA CONNECTOR

Purpose

SAP Java Connector (JCo) is a middleware component that enables you to develop ABAP-compliant components and applications in Java. SAP JCo supports communication with the AS ABAP in both directions: *inbound* (Java calls ABAP) and *outbound* calls (ABAP calls Java).

SAP JCo can basically be integrated with desktop applications and with Web server applications.

The following versions of SAP JCo are available:

- an integrated version that is used with AS Java
- an integrated version that is used within SAP Cloud Platform, for the Neo and Cloud Foundry Environment
- the standalone version that is used to establish a communication between AS ABAP and external Java based business applications (i.e. applications *not* based on SAP's AS Java).

This document exclusively describes the standalone version of JCo 3.1 for the communication with external (*non-SAP*) Java applications.

You can find a description of the integrated SAP JCo as well as further information on the communication between SAP Java applications and the ABAP environment in the SAP Library: <http://help.sap.com>.

Implementation Notes

- For an IDoc-based communication you can additionally use the IDoc Class Library.
- You can access the SAP JCo and IDoc class library installation files at <https://support.sap.com/jco>

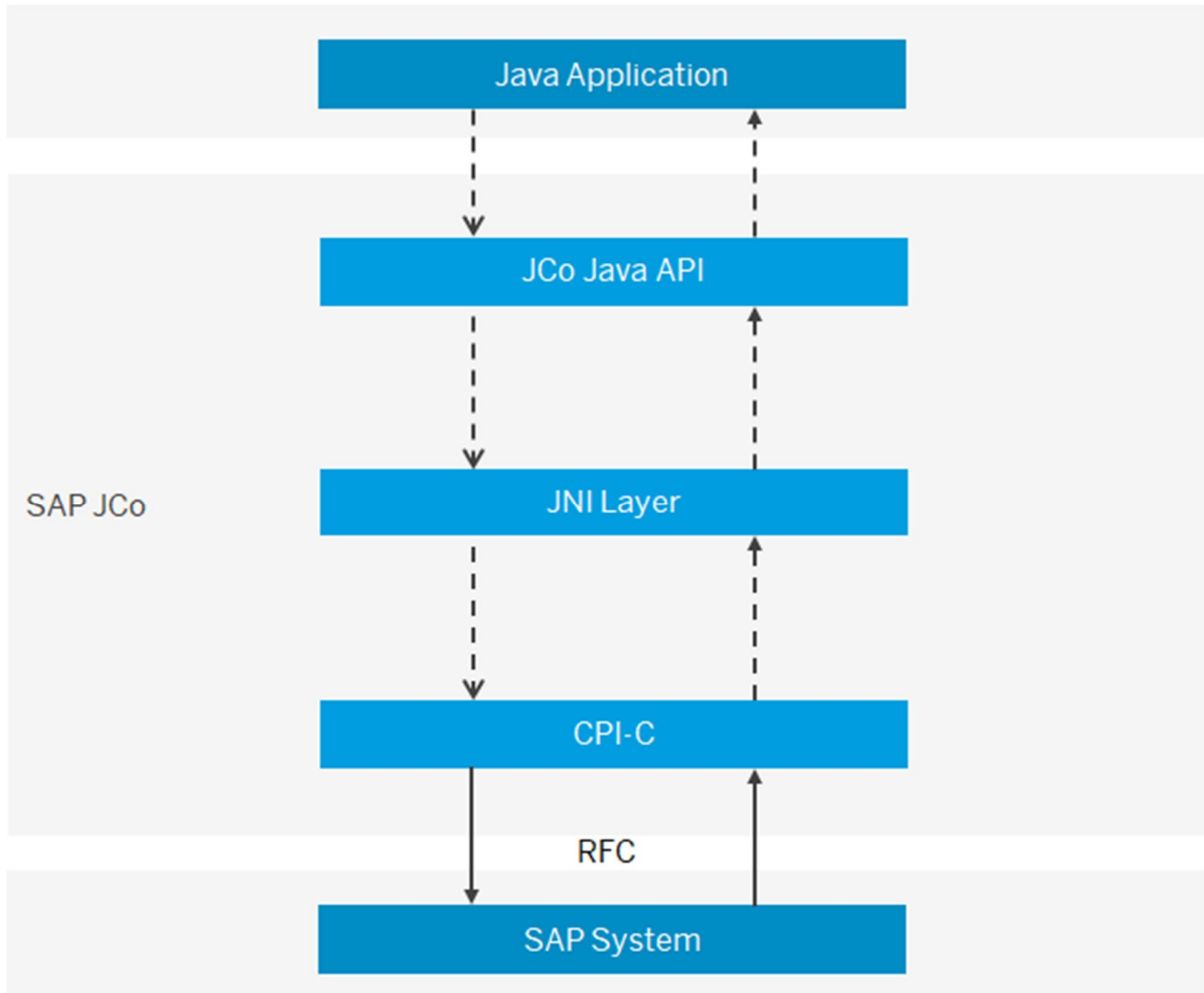
SAP JCo Functions

SAP JCo offers the following functions for creating ABAP-compliant external Java applications:

- SAP JCo is based on JNI (Java Native Interface) that enables access to the CPI-C library.
- It supports SAP (R/3) systems from Release 3.1H upwards, and other SAP components that provide BAPIs or RFMs (Remote-enabled Function Modules).
- A JCo program can execute inbound function calls (Java client calls BAPI or RFM) or receive outbound function calls (ABAP calls external Java Server).
- With SAP JCo, you can use synchronous, transactional, queued and background RFC.
- SAP JCo can be used on different platforms.

SAP JCo Architecture

The following diagram shows the technical schema of data conversion in the SAP JCo (standalone version). Starting from a Java application, a Java method is forwarded via the JCo Java API (Application Programming Interface) to the CPIC layer, where it is converted to an RFC (ABAP) call using the JNI (Java Native Interface) layer and sent to the SAP system. Using the same method in the other direction, an RFC Call is converted to Java and forwarded to the Java application:



SAP JCo Installation

You can download the SAP JCo installation files from SAP Service Marketplace at <https://support.sap.com/jco>.

As the component contains packages as well as native libraries, the native libraries are *platform-dependent*.



Note the additional information on the download page of the SAP Service Marketplace. See SAP note [2786882](#) or detailed information on supported platforms.

Procedure

The following instructions apply for Windows and Linux operating systems. The instructions for the installation of SAP JCo on the individual operating systems are included in the corresponding download files.

1. Create a directory, for example `C:\SAPJCo`, and extract the `sapjco31P_<pl>.zip` file into this directory and expand the ZIP or tgz that is contained in the archive.
2. Make sure that the file `sapjco3.jar` (in the SAP JCo main directory) is contained in the class path for all projects for which you want to use the SAP JCo.

For productive operation, the following files from the archive within `sapjco31P_<pl>.zip` file are required:

- `sapjco3.jar` (Windows and Linux)
- `sapjco3.dll` (Linux: `libsapjco3.so`)



SAP highly recommends that you store `sapjco3.jar` and `sapjco3.dll` (`libsapjco3.so`) in the same directory.

The downloaded .zip file also contains the `javadoc` directory that contains the **Javadoc** for SAP JCo as well as information about the installation, release notes, and the examples. The Javadoc contains an overview of all SAP JCo classes and interfaces that are allowed to be used by applications, together with a detailed description of the corresponding objects, methods, parameters, and fields. Start with the file `intro.html` (`<drive>:\<SAPJCo>\javadoc\ intro.html`) and jump to the other topics in the navigation bar at the top.

SAP JCo Customizing and Environment embedding

After installation SAP JCo can be integrated into the system environment using the package `com.sap.conn.jco.ext`. To do this, implement the corresponding interfaces of the package and register them via the class `com.sap.conn.jco.ext.Environment`.

The following interfaces of the package `com.sap.conn.jco.ext` are especially relevant for JCo embedding into a concrete runtime environment:

Interface	Use
ClientPassportManager	JDSR Passport Manager Interface for client connections to an SAP ABAP application server backend. By default, passports are not sent to the ABAP system.
DestinationDataProvider	Provides the properties for a client connection to a remote SAP system. The default implementation searches <code><destname>.jcoDestination</code> files in the work directory and is thought for an easy start with JCo, but not for productive usage.
ServerDataProvider	Provides the properties for a <code>JCoServer</code> . The default implementation searches <code><servername>.jcoServer</code> files in the work directory and is thought for an easy start with JCo, but not for productive usage.
ServerPassportManager	JDSR Passport Manager Interface for server connections to an SAP application server ABAP backend. By default, passports sent by the ABAP systems are not processed by JCo.
SessionReferenceProvider	Can be implemented by a runtime environment that has a session concept in order to provide JCo with a simple reference to a session. The default implementation considers each thread to be a session.
MessageServerDataProvider	Provides the properties for a client connection to a remote SAP system via a message server. The default implementation searches <code><messageservername>.jcoMessageServer</code> files in the work directory and is thought for an easy start with JCo, but not for productive usage.

Interface	Use
TenantProvider	Can be implemented by a runtime environment that has a multi-tenancy concept to provide information about the current tenant context. By default, JCo assumes to run within a single tenant environment.

For all those runtime embedding capabilities, the highlander principle is in place: There can be only one instance registered with JCo runtime. Hence, deployments should be separated from business application code.

You should always implement the interface `DestinationDataProvider` to optimize data security. If you are using server functionality you should also implement `ServerDataProvider`. These interfaces support the secure storage of critical data.



If application scenarios use stateful call sequences that could span multiple threads, you must implement `SessionReferenceProvider` and register an instance of it using the `Environment` class. This interface connects JCo with session management. As of JCo 3.1, session events are supported for a better integration.

You can find details and an example implementation in the JCo installation directory: `examples/com/sap/conn/jco/examples/advanced/multithreading`

If application scenarios want to isolate runtime objects from each other, you must implement `TenantProvider` and register an instance of it using the `Environment` class. This interface separates JCo-related data for each tenant.

You can find details and an example implementation in the JCo installation directory: `examples/com/sap/conn/jco/examples/advanced/MultiTenantExample.java`.

CLIENT PROGRAMMING

The following section provides an overview of the main elements of client programming when using SAP JCo 3.1 as a standalone component (External (*non*-SAP) Java calls AS ABAP).

More Information

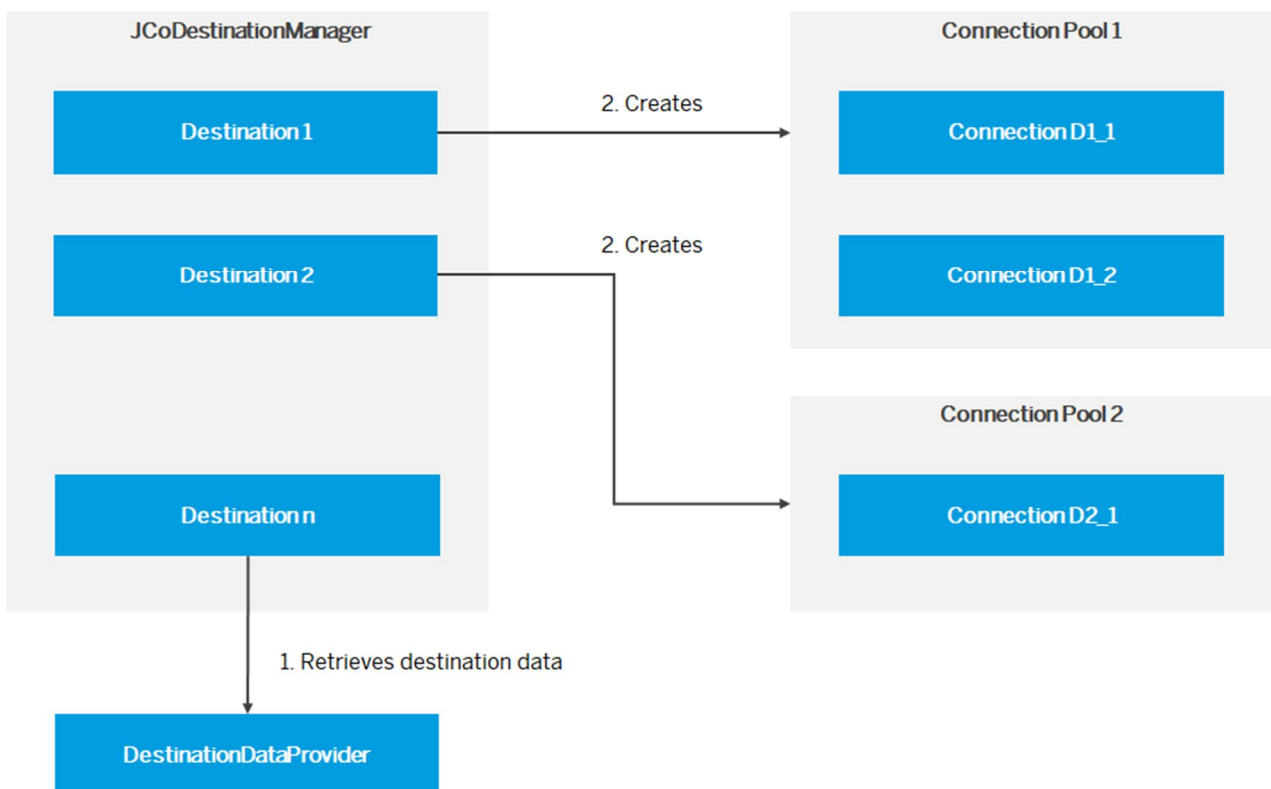
For an overview of all SAP JCo classes, objects, methods, parameters, and fields, see the Javadoc. These HTML files are available in the `javadoc` directory of the SAP JCo installation.

Establishing a Connection to an AS ABAP

Like in JCo 3.0, also in JCo 3.1 the connection setup is not implemented *explicitly* using a single or pooled connection.

Instead, the type of connection is determined only by the connection properties that define a single or pooled connection *implicitly*.

A set of connection parameters (properties) specifies a destination. A destination has a unique name and several connection parameters. The name of a destination usually is a logical name (defined by the application involved). In a productive system, several destinations can point to the same ABAP system.



By specifying the destination name, the corresponding connection (pool) is set up in the background.

The destinations to be called are managed using a destination manager and provided by `JCoDestinationManager`.

The destination manager retrieves the destination properties from `DestinationDataProvider`. `DestinationDataProvider` is an interface that should be implemented depending on the corresponding environment. It gives access to all destinations in the storage it represents and not only a single destination instance.

The implementation delivered by the JCo distribution reads the destination configuration from the file system in the work directory as `<destinationName>.jcoDestination`. However, this procedure is

recommended only for development and testing. Within a productive environment, an adequate and secure solution should be applied.

Defining Destinations

The first step defines destination names and properties.



For this example, the destination configuration is stored in a file that is called by the program. In productive solutions you should avoid this for security reasons and implement a custom destination data provider (see below).

Defining Destinations using the default DestinationDataProvider: file *ABAP_AS.jcoDestination*

```
#for tests only!
jco.client.lang=en
jco.destination.peak_limit=10
jco.client.client=800
jco.client.passwd=alaska
jco.client.user=homo faber
jco.client.sysnr=85
jco.destination.pool_capacity=3
jco.client.ashost=ls4065
```

Defining Destinations using own DestinationDataProvider

```
public static void registerProvider()
{
    DestinationDataProvider my_provider = new MyDestinationDataProvider();
    Environment.registerDestinationDataProvider(my_provider);
}

public static class MyDestinationDataProvider implements DestinationDataProvider
{
    @Override
    public Properties getDestinationProperties(String destinationName)
    {
        // here the properties need to be fetched from a secure storage
        return SomeSecureDatabase.getProperties(destinationName);
    }

    @Override
    public boolean supportsEvents()
    {
        return false;
    }

    @Override
    public void setDestinationDataEventListener(DestinationDataEventListener eventListener)
    {
    }
}
```

Creating Destinations

In this section you find a programming example for structuring a connection to an AS ABAP using the JCO destination concept.

Because the connection type (direct or pool connection) is determined by the destination configuration, it is set implicitly by specifying the destination name. Furthermore, it is not necessary for the application to know, which `DestinationDataProvider` is in place, it simply relies on the destination name and assumes that properties for that destination are available in the storage.



Connection to an AS ABAP

```
JCoDestination dest = JCoDestinationManager.getDestination(DESTINATION_NAME);
System.out.println("Attributes: " + dest.getAttributes());
```

Custom Destinations (optional)

Besides the described destination concept, you can also create a `JCoCustomDestination` from an existing destination instance. Some properties such as user data can be changed there on the fly without obtaining a new destination instance from `JCoDestinationManager`. Thus, e.g. a base destination could be used and for logging on users, credentials are used that are based on the input provided in a dialog.



Custom destination

```
JCoCustomDestination cust_dest = dest.createCustomDestination();
UserData usData = cust_dest.getUserLogonData();
usData.setUser("HUGO");
usData.setPassword("HUGO");
```

Executing Function Modules in an SAP System

Executing a Stateless Call

In the following example a function is executed by calling a function module without keeping a state on the ABAP side.



Executing Simple Functions

You want to call e.g. function `RFC_PING`. You call a destination and the corresponding function via the repository.

```
JCoFunction function = dest.getRepository().getFunction("RFC_PING");
if (function == null)
    throw new RuntimeException("RFC_PING not found in SAP.");

try
{
    function.execute(dest);
}
catch (AbapException e)
{
    System.out.println(e);
}
```

Executing a Stateful Call Sequence

You need a stateful connection to an AS ABAP if you want to execute multiple function calls in the same session (in the same context).

Therefore, you must declare a *stateful* connection explicitly.

JCo Client: Stateful Connection

For a *stateful* connection, use the statements `JCoContext.begin(destination)` and `JCoContext.end(destination)`. All function modules executed in between will get into the very same ABAP system session and state on ABAP system side will be kept from one call to the next one. The state will be released after `JCoContext.end(destination)` is called.

```
JCoFunction bapiFunction1 = ...
JCoFunction bapiFunction2 = ...
JCoFunction bapiTransactionCommit = ...
JCoFunction bapiTransactionRollback = ...

try
{
    JCoContext.begin(dest);
    try
    {
        bapiFunction1.execute(dest);
        bapiFunction2.execute(dest);
        bapiTransactionCommit.execute(dest);
    }
    catch (AbapException ex)
    {
        bapiTransactionRollback.execute(dest);
    }
}
catch (JCoException ex)
{
}
finally
{
    JCoContext.end(dest);
}
```

Starting a SAP GUI

When calling a function module in the SAP system it may be necessary to start an SAP GUI on your client.



Some (older) BAPIs need this, because they try to send screen output to the client while executing.

If you want to start an SAP GUI on an external client, your SAP backend system must meet some requirements. You will find detailed information in SAP note **1258724**.

To start an SAP GUI from your client program, proceed as follows:

Windows:

Set the *property* `USE_SAPGUI` to 1 (visible) or 2 (hidden).



Prerequisite: You have installed the Windows SAP GUI on your system.

Possible values are:

0: no SAPGUI (default)

1: attach a visible SAPGUI.

2: attach a "hidden" SAPGUI, which just receives and ignores the screen output.

Unix:

For Unix systems, a Java SAP GUI is required. In addition, the environment variable `SAPGUI` needs to be set.

Set the environment variable via:

```
setenv SAPGUI <path to SAPGUI start script> (tcsh) or  
export SAPGUI=<path to SAPGUI start script> (bash)
```

For Mac OS the path name could look like this:

```
"/Applications/SAP Clients/SAPGUI 7.60rev7.3/SAPGUI  
7.60rev7.3.app/Contents/MacOS/SAPGUI"
```

For Linux and other Unix systems the following path would be valid:

```
/opt/SAPClients/SAPGUI7.60rev7.3/bin/sapgui
```



You must not add *any* parameter after the script name.



Start the JCo application from the *same* shell to propagate the environment variable to the program.

Using Function Module Parameters

Besides just calling a function module, an application typically wants to set and get import, export and table parameters.

Scalar Parameters



Using function parameters

This example calls function `STFC_CONNECTION` using the input parameter `REQUTEXT`.

1. To access the import parameter list, use `getImportParameterList()`.
2. The value of the scalar parameter is set using `setValue(<name>, <value>)`, in which first the name, and then the value is entered. There are many different versions of `setValue(<name>, <value>)` in SAP JCo to support various data type conversions directly.
3. SAP JCo converts the values and transfers them to the data type that is assigned to the field. If an error occurs during the conversion, an exception is thrown. The method `setValue(<name>, <value>)` is also available in `JCoStructure` and `JCoTable`, which both inherit this method from `JCoRecord`. You can therefore set values for structure fields and fields in a row of a table in the very same way (see for more details in the subsection).

4. After execution, the export parameter list can be accessed by `getExportParameterList()`.
5. Like `setValue(<name>, <value>)` you can call `getValue(<name>)` in order to retrieve the content as the Java type that is fitting best to the ABAP type.
6. The method `setValue(<index>, <value>)` is useful in `JCoStructure` and `JCoTable`, as the position is clear when accessing concrete fields in those cases.



If you know the data type of the respective field, you can also call type-specific *get* methods (see in section ["Using suitable getter methods"](#)).



If a parameter should never be sent or fetched, you can call `setActive()` in which first the name, and then the new state is entered (see [here](#)).

```
JCoFunction function = dest.getRepository().getFunction("STFC_CONNECTION");
if (function == null)
    throw new RuntimeException("STFC_CONNECTION not found in SAP.");

function.getImportParameterList().setValue("REQUTEXT", "Hello SAP");

try
{
    function.execute(dest);
}
catch (AbapException e)
{
    System.out.println(e);
    return;
}

System.out.println("STFC_CONNECTION finished:");
System.out.println(" Echo: " + function.getExportParameterList().getString("ECHOTEXT"));
System.out.println(" Response: " + function.getExportParameterList().getString("RESPTEXT"));
```

Structure & Table Parameters

Reading a Table

In the next example the function module `BAPI_COMPANYCODE_GETLIST` is called and a table of all the company codes is displayed. This RFM does not contain any import parameters.

1. First, the structure `RETURN` is checked. The content of each field of a structure can be accessed similarly with `getValue()` methods. For flat and char-like structures/tables `getString()` can be called, which returns all fields of the current row at once.
2. Second, if the returned type fits, the table is accessed with `getTableParameterList()`.
3. Within this list, access the actual table (`getTable(<name>)`). The interface `JCoTable` contains all methods that are available for `JCoStructure`, together with additional methods for navigation in a table. A table can have an arbitrary number of rows or can also have no rows at all. The code snippet below shows navigation with the method `setRow(<row>)`, in which the current row pointer is moved to every row in the table in turn. The method `getNumRows()` specifies how many rows exist in total. Instead of using `setRow(<row>)`, you can also use the method `nextRow()`, which moves the row pointer by one as displayed in the lower section of the code snippet.



Accessing a Structure and Table

```

JCoFunction function = dest.getRepository().getFunction("BAPI_COMPANYCODE_GETLIST");
if (function == null)
    throw new RuntimeException("BAPI_COMPANYCODE_GETLIST not found in SAP.");

try
{
    function.execute(dest);
}
catch (AbapException e)
{
    System.out.println(e);
    return;
}

JCoStructure returnStructure = function.getExportParameterList().getStructure("RETURN");
if (! (returnStructure.getString("TYPE").equals("") || returnStructure.getString("TYPE").equals("S")))
    throw new RuntimeException(returnStructure.getString("MESSAGE"));

JCoTable 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"));
}

```



Accessing a Table (Variant)

```

[...]
codes.firstRow();
do
{
    System.out.println(codes.getString("COMP_CODE") + '\t' + codes.getString("COMP_NAME"));
}
while (codes.nextRow());

```

Modifying a Table

In many applications it is not enough to be able to access table fields or navigate through a table, you also often need to add or delete rows. SAP JCo provides methods that enable you to do this. Normally, you add rows for table parameters that are sent to the AS ABAP (for example, adding items to a customer order).

The following table summarizes the JCoTable methods that are not contained in JCoStructure.

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.

Method	Description
void appendRow()	Adds one row at the end of the table.
void appendRows(int num_rows)	Adds multiple rows at the end of the table (better performance than calling <code>appendRow()</code> multiple times).
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.

Modifying a Table

In this example we fill a table with 100 rows having all the same company code. The respective `set` methods will convert the value of the passed object to the field in the current row.

```
[...]
codes.appendRows(100);
codes.firstRow();
do
{
    codes.setValue("COMP_CODE", "YYYY"); // or setString
}
while (codes.nextRow());
```

Using Multi-Threading

Most applications run in several threads. JCo classes like repository or destination are synchronized and may be used from several threads at the same time.

As long as all stateful call sequences are executed within the same thread (i.e. in the same thread the first call has been performed in), the default `SessionReferenceProvider` implementation provided by JCo is sufficient.

However, within an application server this is frequently not the case, and a running session may change the thread while being processed.

In this case the environment must provide an adequate implementation of `SessionReferenceProvider`. This enables the JCo runtime to assign the function module executions in different threads to a unique session and can thus decide which connection needs to be chosen in order to end up in the correct ABAP session for all of the stateful function module executions.



In a multi-thread environment, the use of container objects (for example, `JCoTable` objects) from different threads must be implemented carefully. Note further, that it is not possible to make multiple concurrent SAP calls for the very same connection. Hence, if a `JCoContext` was started within a session, you cannot execute multiple requests in concurrent threads belonging to this session at the very time. When trying this a `JCoException` with group `JCO_ERROR_CONCURRENT_CALL` will be thrown in the second thread trying to perform an execution.

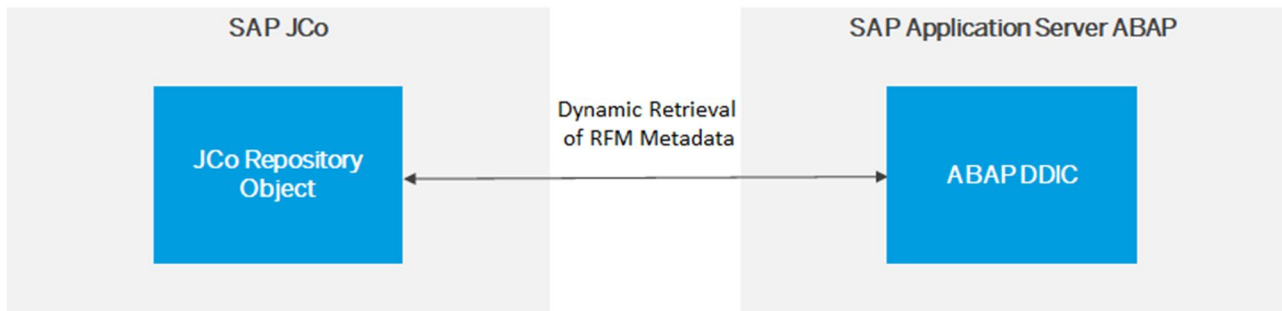
Please refer to the example in `examples/advanced/multithreading` to see how a stateless as well as stateful connection is used.

SAP JCo Repository

The SAP Java Connector must be able to access the metadata of all Remote Function Modules (RFMs) that are to be used by a Java client. A `JCoRepository` object is created to do this. The current metadata for the RFMs is retrieved either dynamically from the AS ABAP at runtime (recommended), from a previously serialized instance, or hard-coded.



You must only create the `JCoRepository` object in the case of hard-coded metadata. This step is automatically executed internally by JCo when retrieving metadata.



Obtaining a Repository

For the JCo Repository, you need the following interfaces:

- `JCoRepository`: Contains the runtime metadata of the RFMs.
- `JCoFunctionTemplate`: Contains the metadata for an RFM.
- `JCoFunction`: Represents an RFM with all its corresponding parameters.
- `JCoListMetaData`: Defines the metadata for parameter lists of an RFM.
- `JCoRecordMetaData`: Defines the metadata for structures and tables.

Make sure that the user ID for the Repository has all the required authorizations for accessing the metadata of the AS ABAP. (See SAP note [460089](#).)



The user ID used for repository queries is specified on the corresponding destination.



You must lock the metadata before using them to avoid any unintended changes. For repositories with dynamic lookups the lock happens automatically

Getting the JCoRepository from a JCoDestination

```
JCoRepository mRepository;  
mRepository = dest.getRepository();
```

Creating JCoFunction Objects

To create a JCoFunction object, you can use one of the following two options:

- Execute the method `getFunction()` for the repository instance, or
- Execute the method `getFunctionTemplate()` for the repository instance and then `getFunction()` on the retrieved template.

In addition to the fitting metadata, a function object also contains the current parameters for executing the RFMs. The relationship between a function template and a function in SAP JCo is like that between a class and an object in Java. The code displayed above encapsulates the creation of a function object.



SAP recommends that you create a *new* function object for each individual execution. By doing so you can ensure that the parameters do not contain any elements from previous calls, which is very likely undesired.

Mapping of ABAP and Java Data Types

A data structure is made up of individual fields, and each field is assigned to a data type. Because ABAP uses different data types than Java, it is necessary to create a link between these data types (*mapping*). The table displayed below shows the different data types in ABAP and Java and their mapping when using the `getValue()` method.

ABAP Type	Description	Java Data Type	Java Type Code
c	Character	String	JCoMetadata.TYPE_CHAR
n	Numerical Character	String	JCoMetadata.TYPE_NUM
x	Binary Data	Byte array	JCoMetadata.TYPE_BYTE
p	Binary Coded Decimal	BigDecimal	JCoMetadata.TYPE_BCD
int8	8-byte Integer	long	JCoMetadata.TYPE_INT8
i	4-byte Integer	Int	JCoMetadata.TYPE_INT
b	1-byte Integer	Int	JCoMetadata.TYPE_INT1
s	2-byte Integer	Int	JCoMetadata.TYPE_INT2
f	Float	double	JCoMetadata.TYPE_FLOAT
d	Date	Date	JCoMetadata.TYPE_DATE
t	Time	Date	JCoMetadata.TYPE_TIME

ABAP Type	Description	Java Data Type	Java Type Code
utclong	Timestamp	String	JCoMetadata.TYPE_UTCLONG
decfloat16	Decimal floating point 8 bytes (IEEE 754r)	BigDecimal	JCoMetadata.TYPE_DECF16
decfloat34	Decimal floating point 16 bytes (IEEE 754r)	BigDecimal	JCoMetadata.TYPE_DECF34
string	String (variable length)	String	JCoMetadata.TYPE_STRING
xstring	Raw String (variable length)	Byte array	JCoMetadata.TYPE_XSTRING

In most cases, handling of data types does not represent a problem. However, the ABAP data types for date and time have some special features. ABAP has two different data types for processing date and time information:

- ABAP data type `T` is a 6-char string with the format HHMMSS
- ABAP data type `D` is an 8-char string with the format YYYYMMDD

Both data types are used in RFMs (including BAPIs). When a BAPI uses a time stamp two fields are used, one of type `D` and one of type `T`.

Java, however, only uses *one* class (`Date`) to represent both date and time information. In Java, a time stamp can therefore be displayed in one variable.

SAP JCo automatically performs the conversion between ABAP and Java data types. Fields of the ABAP data types `D` and `T` are represented as Java `Date` objects, whereby the part of the `Date` object that is not used retains its default value. Java developers must distinguish between whether a field contains an ABAP date value or an ABAP time value.

Using suitable getter methods

The interface `JCoStructure` contains type-specific getter methods such as `getString()` for the data type string. Applications normally use the appropriate getter method; you can of course use another getter method: SAP JCo then tries to convert the field content to a relevant data type. If this conversion is unsuccessful (for example, if a string field contains the value "abcd", and you call `getDate()`), an exception is thrown.

The table below lists all the type-specific getter methods. The method `getValue()` can be used to call the content of a field generically. This method returns a Java object.

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

JCo Type Code	JCo Access Method
JCoMetadata.TYPE_STRING	String getString()
JCoMetadata.TYPE_XSTRING	byte[] getByteArray()
JCoMetadata.TYPE_DECF16	BigDecimal getBigDecimal()
JCoMetadata.TYPE_DECF34	BigDecimal getBigDecimal()
JCoMetadata.TYPE_INT8	long getLong()
JCoMetadata.TYPE_UTCLONG	String getString



It is not required to use exactly the methods mentioned above for a concrete datatype. JCo will try to convert any value to the Java type and if this is not possible, `ConversionException` will be thrown.

Generic processing of fields

Fields can occur in different contexts: Structures and table rows contain fields, and scalar parameters are fields. The previous sections describe how each interface supports methods for accessing or changing the content of a field. Because fields in different contexts have some common features, SAP JCo provides the `JCoField` interface, which enables generic editing of fields.

This is an advantage for some special scenarios. However, the `JCoField` classes always produce a certain overhead as they are designed as wrapper classes. Hence, `JCoRecord` classes like `JCoStructure` or `JCoParameterList` offer `getFieldIterator()` as well as `iterator()` methods.

With the iterator you can access the single fields. `JCoField` can also be obtained from `getField()` from `JCoRecord`.

The `JCoField` interface itself contains all the getter and setter methods described earlier. This level of abstraction can be very useful if you want to create generic methods for editing fields, irrespective of the origin of the fields. A field of the `JCoField` interface has metadata such as:

- Name (method `getName()`)
- Description (method `getDescription()`)
- Data type (method `getType()`)
- Length (method `getLength()`), and
- Number of decimal places (method `getDecimals()`)

A field can also contain extended metadata that you can access using method `getExtendedFieldMetaData()`.



As already denoted above using `JCoField` only makes sense in very special cases. Usually, it is better to use the data containers and the corresponding `set` and `get` methods directly, in particular, with regards to performance they are superior to `JCoField`.

Optimizing performance by deactivating parameters

The previous sections have described the principles for working with parameters. You know how to access a structure, a table, and scalar parameters. This section contains further advice for optimizing performance:



Many BAPIs have a large number of parameters, not all of which are used in an application. You can prevent SAP JCo to retrieve or send unused parameters. To do this, simply declare the parameter as *inactive*, as displayed in the code snippet below. This is particularly effective for larger tables that are returned by the application or if the calculation of a certain return value is very time-consuming.

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

JSON Import/Export of JCoRecord

To store and load different content of a `JCoRecord` externally, you can convert it with `toJSON()` into a JSON string and read it again with `fromJSON()`. This is useful in solutions, in which JCo is interacting in a Servlet container with an HTML5 based Browser application to avoid boilerplate code.

Connection check

Using the property `jco.destination.pool_check_connection` in the `DestinationDataProvider`, you can now check the availability of a connection before the actual RFM call. It is possible to recognize corrupted connections, and avoid exceptions passed to applications if connectivity is working in principle.

The state of this property can be checked with `isPooledConnectionChecked()` in `JCoDestination`.

Fast RFC serialization

JCo 3.1 offers to transmit data in a more efficient way using the fast RFC serialization, which is technically based on a column-based serialization approach. Compared to older serialization mechanisms, data is compressed in a highly efficient manner and can reduce the overall network load significantly. To activate it, the destination which should use it, must have the following properties:

- `jco.client.serialization_format` – valid values are **rowBased** or **columnBased**. For fast serialization **columnBased** must be set. The default serialization is **rowBased**, which will serialize the data with the previous approach.
- `jco.client.network` – valid values are **LAN** or **WAN** (relevant for fast serialization only). In the **WAN** case, a slower, but better compression algorithm is used and in addition, the data is analyzed for further compression options. For **LAN** a very fast compression algorithm is used, and data analysis is done only at a very basic level. Hence, the compression ratio is not as good in that case, but in this case the transfer time is considered not that dominant like for **WAN**. Choose this value depending on the network quality between JCo and your target system to optimize the performance. The default setting is **LAN**.

Sticky load-balanced destinations

For load balancing destinations you can now “stick” to a given application server which is once chosen as soon as the first connection has been opened this destination. This means, that all successor connections will connect to that same application server afterwards. To activate this behavior for a given destination, the property `jco.client.sticky=1` must be provided. By default, load balancing remains to be not sticky.

Retrieving message server and logon group information

JCo 3.1 introduces new APIs to retrieve information, such as logon groups, from a message server. Firstly, for connecting to a message server, a `MessageServerDataProvider` must be implemented and registered by the `Environment`. Like `DestinationDataProvider`, the properties identifying a concrete message server must be retrievable:

- `jco.message_server.host` – the host of the message server
- `jco.message_server.service` – the message server service to use for connecting to it, or
- `jco.message_server.system_id` – the system ID of the system the message server belongs to



You can now create a message server from `JCoMessageServerFactory` and retrieve its system information:

```
public static void main(String[] args)
    throws MessageServerQueryException, JCoException
{
    MyMessageServerProvider msp = new MyMessageServerProvider();
    Environment.registerMessageServerDataProvider(msp);

    JCoMessageServer messageServer = JCoMessageServerFactory.getMessageServer("test");

    // get information
    for (JCoLogonGroup logonGroup : messageServer.getLogonGroups())
        System.out.println(logonGroup.getName());

    for (JCoApplicationServer appServer : messageServer.getApplicationServers())
    {
        System.out.println(appServer.getHostName());
        System.out.println(appServer.getInstanceNumber());
        // [...]
    }

    Environment.unregisterMessageServerDataProvider(msp);
}

private static class MyMessageServerProvider implements MessageServerDataProvider
{
    @Override
    public Properties getMessageServerProperties(String messageServerName)
    {
        if (messageServerName.equals("test"))
        {
            Properties props = new Properties();
            props.put(MessageServerDataProvider.MESSAGE_SERVER_HOST, "localhost");
            props.put(MessageServerDataProvider.SYSTEMID, "ALX");
            return props;
        }
        else
        {
            return null;
        }
    }

    @Override
    public void setMessageServerDataEventListener(MessageServerDataEventListener eventListener)
    {
    }

    @Override
    public boolean supportsEvents()
    {
        return false;
    }
}
```

Exception Handling

For exception handling, you need the following classes:

- **JCoException**: root class for exceptions.
 - If the group `JCO_ERROR_SYSTEM_FAILURE` is used by the `JCoException`, then there are multiple causes possible: It could be an ABAP message (will provide message related info in addition), a short dump of the work process, or a core dump. Check in the ABAP system for the reason, the runtime error monitor (transaction ST22) could be a good entry point. In this case, the connection will be closed and if a connection was set to stateful, the state will be lost.
 - **ABAPException**: subclass for exceptions thrown by an RFM. The connection remains open and can still be used.

An `AbapException` occurs if the ABAP code that you have called with JCo throws a classic exception. Check in the ABAP code for the reason.

- **JCoRuntimeException**: root class for runtime exceptions.
 - **ConversionException**: special class for conversion errors.

A `ConversionException` is thrown, if the code calls a getter or setter method that requires a conversion, and this conversion fails.

- **XMLParserException**: used for errors in the XML parser.

`XMLParserException` is thrown by the XML parser, if invalid or corrupt data arrives for a complex (deeply nested) parameter.

All exceptions thrown by SAP JCo are subclasses of `JCoException` or `JCoRuntimeException`.

An exception that is expected to be thrown as a 'checked' exception, is thrown as a subclass of `JCoException` or directly of this type. However, when using these 'normal' exceptions the compiler requests always to decide whether an exception is to be handled explicitly with a try-catch-finally block or if it needs to be added to the `throws clause`. *JCoRuntimeExceptions* are used whenever the error can happen but is typically not expected. In this case, the compiler will not enforce handling, but the code should handle it at some place in the invocation hierarchy.



Exception Handling – Option 1

```
public void executeFunction(JCoFunction function, JCoDestination dest)
    throws JCoException
{
    function.execute(dest);
}
```




Exception Handling – Option 2

```
public void executeFunction(JCoFunction function, JCoDestination dest)
{
    Try
    {
        function.execute(dest);
    }
    catch (AbapException ex)
    {
        // clause 1
        if (!ex.getKey().equalsIgnoreCase("RAISE_EXCEPTION"))
        {
            // ...
        }
    }
    catch (JCoException ex)
    {
        // clause 2
        logSystemFailure(ex.getMessage());
    }
    catch (JCoRuntimeException ex)
    {
        // clause 3
    }
    catch (Exception ex)
    {
        // clause 4
    }
}
```

There are four *catch* clauses:

- In the first catch clause you use the method `getKey()`, to access the exception string returned by the SAP system. In this way you can distinguish between different ABAP exceptions that you want to handle in a specific way, and all others that are handled generically. Because all exception strings are defined in SAP, you already know them in advance.
- The second and third catch clauses refer to all other JCo-related problems. These include conversion errors and other errors that occur in SAP JCo. With e.g. `getMessage()` you can access the exception text. But note that also errors in the remote system are forwarded via this exception class and are not an error within JCo.
- The fourth clause refers to all other exceptions that may have occurred in your code.



Note that this is also an example description. Depending on the concrete requirements of your code it may be required to adjust the error handling.

More Information

You can find a detailed description of the individual exception classes in JCo Javadoc in the installation directory `javadoc`.

SERVER PROGRAMMING

The following section explains how you can write your own JCo server program, if you use the standalone version of the SAP JCo.

A JCo server program implements functions that are called by an ABAP backend. The JCo server program is registered via one (or more) SAP Gateways and waits for inbound RFC calls.

1. An RFC server program registers itself under a program ID to an SAP gateway (not for a specific SAP system).
2. If an RFC call is passed on from any SAP system to this SAP gateway with the option "Connection with a registered program" (with the same program ID), a connection is opened to the corresponding JCo server program.
3. Once a remote-enabled function module has been executed, the JCo Server waits for further RFC calls from the same or other SAP systems.
4. If an RFC connection is interrupted or terminated, the JCo server automatically registers itself again on the same SAP gateway under the same program ID.

Prerequisites

- You are using the standalone version of SAP JCo.
- You have defined a destination with connection type T (TCP/IP connection) in the SAP system via transaction SM59.
- You have chosen the registration mode ("Registered server program" option under the "Technical settings" tab) for this destination.
- The gateway information in the destination matches the corresponding parameters used for starting the registered RFC server program.

Calling an RFC Server from AS ABAP

This section provides an example of how you can establish an RFC connection from the SAP system to an RFC server program.

To send a call from an ABAP system, the ABAP program uses the addition `DESTINATION "NAME"` to the command `CALL FUNCTION`.

www.sap.com/contactsap

© 2019 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

The information contained herein may be changed without prior notice. Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platform directions and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, and they should not be relied upon in making purchasing decisions.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies. See www.sap.com/copyright for additional trademark information and notices.

THE BEST RUN





ABAP Program for RFC Inbound Connection

In transaction `SE38`, create a report with the following coding:

```
DATA: REQTEXT LIKE SY-LISEL,
      RESPTXT LIKE SY-LISEL,
      ECHOTEXT LIKE SY-LISEL.

DATA: RFCDEST like rfcdes-rfcdest VALUE 'NONE'.
DATA: RFC_MESS(128).

REQTEXT = 'HELLO WORLD'.
RFCDEST = 'JCOSEVER01'. "corresponds to the destination name defined in the SM59

CALL FUNCTION 'STFC_CONNECTION'
  DESTINATION RFCDEST
  EXPORTING
    REQTEXT = REQTEXT
  IMPORTING
    RESPTXT = RESPTXT
    ECHOTEXT = ECHOTEXT
  EXCEPTIONS
    SYSTEM_FAILURE = 1 MESSAGE RFC_MESS
    COMMUNICATION_FAILURE = 2 MESSAGE RFC_MESS.

IF SY-SUBRC NE 0.
  WRITE: / 'Call STFC_CONNECTION          SY-SUBRC = ', SY-SUBRC.
  WRITE: / RFC_MESS.
ENDIF.
```

Java Program for Starting an RFC Server

In the next step, you can write a Java program that registers a server connection at an SAP gateway.

To do this, you must:

- Implement the `JCoServerFunctionHandler` and the coding to be executed when the call is received.
- Define and provide the server properties for the `JCoServer`.
- Create an instance for your `JCoServer` implementation and start it with `start()`.

Function Handler



This function prints out some connection information and responds by copying the import text `REQTEXT` to `ECHOTEXT` and adds a `RESPTXT`.

```
public class StfcConnecti onHandl er implements JCoServerFuncti onHandl er
{
    @Override
    public void handl eRequest(JCoServerContext serverCtx, JCoFuncti on functi on)
    {
        System.out.println("call: " + functi on.getName());
        System.out.println("Connecti onId: " + serverCtx.getConnecti onID());
        System.out.println("gwhost: " + serverCtx.getServer().getGatewayHost());
        // [...]

        final String reqtext = functi on.getImportParameterLi st().getStri ng("REQTEXT");
        System.out.println("req text: " + reqtext);
        functi on.getExportParameterLi st().setVal ue("ECHOTEXT", reqtext);
        functi on.getExportParameterLi st().setVal ue("RESPTXT", "Hel lo Worl d");
    }
}
```

Defining Server Properties

The first step defines a server name and properties.



For this example, the server configuration is stored in a file that is called by the program. In practice you should avoid this for security reasons and implement an individual server data provider (see below).

Defining Server Properties using the default `ServerDataProvider`: file `SERVER.jcoServer`

```
#for tests only!
jco.server.connection_count=2
jco.server.gwhost=binmain
jco.server.progid=JCO_SERVER
jco.server.gwserv=sapgw53
jco.server.repository_destination=ABAP_AS_WITH_POOL
```

Defining Server Properties using an own `ServerDataProvider`

```
public static void registerProvider()
{
    ServerDataProvider my_provider = new MyServerDataProvider();
    Environment.registerServerDataProvider(my_provider);
}

public static class MyServerDataProvider implements ServerDataProvider
{
    @Override
    public Properties getServerProperties(String serverName) throws DataProviderException
    {
        // here the properties need to be fetched from a secure storage
        return SomeSecureDatabase.getProperties(serverName);
    }

    @Override
    public boolean supportsEvents()
    {
        return false;
    }

    @Override
    public void setServerDataEventListener(ServerDataEventListener eventListener)
    {
    }
}
```

Setting up a JCo Server



Now a server can be created and started.

```
JCoServer server;
try
{
    server = JCoServerFactory.getServer(SERVER_NAME);
}
catch (JCoException ex)
{
    throw new RuntimeException("Unable to create server " + SERVER_NAME + ", because of " +
        ex.getMessage(), ex);
}

JCoServerFunctionHandler stfcConnectionHandler = new StfcConnectionHandler();
DefaultServerHandlerFactory functionHandlerFactory =
    new DefaultServerHandlerFactory.FunctionHandlerFactory();
factory.registerHandler("STFC_CONNECTION", stfcConnectionHandler);
server.setCallHandlerFactory(factory);

server.start();
```



The `DefaultServerHandlerFactory` is enough for most of the use cases and should be preferred if possible. In case you need to offer a more sophisticated implementation for your server, it might become necessary to implement a custom `JCoServerFunctionCallHandlerFactory`. In this case, the implementation needs to return the fitting `FunctionCallHandler` for the concrete invocation.

Implementing an Exception Listener

Whenever errors occur, the JCo throws an *exception*. All exceptions that occur within JCo runtime are passed on to the registered *Exception* and *Error Listener*.



This applies only to exceptions happening in the JCo runtime, e.g. network failures, partner system down, data conversion failures etc. Exceptions that the application coding throws itself inside the `handleRequest()` method, are not passed on to those Listeners. If you want to monitor those as well, you need to do so at the point where you throw them.

To define this listener, create a class that implements `JCoServerExceptionHandler` and `JCoServerErrorListener`:



Exception and Error Listener

```

public class MyThrowableListener
    implements JCoServerErrorListener, JCoServerExceptionListener
{
    public void serverErrorOccurred(JCoServer jcoServer, String connectionId,
                                    JCoServerContextInfo serverCtx, Error error)
    {
        System.out.println(">>> Error occurred on " + jcoServer.getProgramID() + " connection " +
                           connectionId);
        error.printStackTrace();
    }

    public void serverExceptionOccurred(JCoServer jcoServer, String connectionId,
                                        JCoServerContextInfo serverCtx, Exception error)
    {
        System.out.println(">>> Exception occurred on " + jcoServer.getProgramID() +
                           " connection " + connectionId);
        error.printStackTrace();
    }
}

```

Register the listener class with `addServerErrorListener()` and `addServerExceptionListener()`:



Registering the Listener Class

```

MyThrowableListener eListener = new MyThrowableListener();
server.addServerErrorListener(eListener);
server.addServerExceptionListener(eListener);

```

Monitoring Server Connections

The `JCoServerStateChangedListener` class enables you to monitor state changes of the server:



Monitoring Server Connections

```

public class MyStateChangedListener implements JCoServerStateChangedListener
{
    // see JCoServerState class for details
    public void serverStateChangeOccurred(JCoServer server, JCoServerState oldState,
                                          JCoServerState newState)
    {
        // Details for connections managed by a server instance are available via
        // JCoServerMonitor
        System.out.println("Server state changed from " + oldState.toString() + " to " +
                           newState.toString() + " on server with program id " + server.getProgramID());
    }
}

```

Register the listener class with the API `JCoServer.addServerStateChangedListener()`:



Registering the Listener Class

```

MyStateChangedListener sListener = new MyStateChangedListener();
server.addServerStateChangedListener(sListener);

```

Processing an ABAP Call

The application code that processes an RFC call, should in general trigger only ABAP exceptions and ABAP messages. All other (runtime) exceptions that are thrown by `handleRequest()` are reported back to the SAP system as a `SYSTEM_FAILURE`.



Throwing an ABAP Exception or Message

```
@Override
public void handleRequest(JCoServerContext serverCtx, JCoFunction function)
    throws AbapException
{
    // processing the request
    // [...]
    // throw a classic function module exception, would be done using RAISE in ABAP
    if (reasonForABAPException)
        throw new AbapException("EXCEPTION_KEY", "descriptive text");
    // throw a message exception, would be done using MESSAGE ... RAISING in ABAP
    if (reasonForABAPMessage)
        throw new AbapException("EXCEPTION_KEY", "AB", 'E', "007",
            new String[] {"parameter 1", "parameter 2"});
}
```

As mentioned before, the exception and error listeners are not informed about this exception, as it is thrown by application code in `handleRequest()`.

Processing a tRFC/qRFC /bgRFC Call

SAP JCo can also process tRFC/qRFC and bgRFC calls of type T (transactional) and Q (queued). The processing logic within JCo is the same for tRFC/qRFC and bgRFC.



If the following ABAP statement is executed:

```
CALL FUNCTION 'STFC_CONNECTION'
  IN BACKGROUND TASK
  DESTINATION 'RFCDEST'.

(bgRFC: IN BACKGROUND UNIT)
```

tRFC/qRFC/bgRFC processing takes place.

The following section describes the use of tRFC/qRFC and bgRFC.

tRFC/qRFC Calls

For the use of tRFC and qRFC, the following call sequence is triggered:



Processing requires a custom implementation of `JCoServerTIDHandler`.

1. `boolean checkTID(String tid) // on your implementation of JCoServerTIDHandler`



At this point, the application must be informed that the next call is to be processed with this TID. The application must return the value `true` in order to signal that execution is possible. The value `false` should be returned if a unit with this ID has already been processed. In case of any errors an exception should be thrown, which will abort the unit processing. Ideally, the TID is stored in a permanent storage.

2. All actual function modules are invoked with `handleRequest()`

3. `commit(String tid)` or `rollback(String tid)`



Depending on the result of the function invocations, if none of the `handleRequest()` invocations has thrown any exceptions, `commit` is called. Otherwise `rollback` is called. It is important to ensure atomic execution of the transactions, i.e. the transaction is done completely or not at all, as well as exactly once semantics. The application needs to guarantee this behavior, e.g. if TIDs are stored in a DB, a respective commit or rollback should be performed. Also, if the application code in the function performs such logic, it needs to commit altogether or rollback all changes made during processing of the function modules that are contained in the unit.

4. `confirmTID(String tid)`



At this point, the application is informed that the client knows that the request has ended for the specified TID. The application can now release all resources associated with this TID.

Under certain circumstances, this call might be received in a different *Listener* or, if problems arise, may not be received in the ABAP backend system at all.

Example implementations for tRFC/qRFC calls as well as bgRFC can be found in `examples/com/sap/conn/jco/examples/server/advanced/trfc` and `examples/com/sap/conn/jco/examples/server/advanced/bgrfc`.

bgRFC Calls

For bgRFC, the process is similar. The difference is, that the ID is passed as an instance of `JCoUnitIdentifier` instead of a string. Furthermore, the application needs to implement `getFunctionUnitState(unitIdentifier)` which returns the current `JCoFunctionUnitState` and for that must track the current state of each unit.

Stateful Server Calls

A `JCoServer` function module can be implemented in a way that it can execute stateful call sequences instead of being stateless. In this case, a call to `JCoContext.setStateful()` with parameter value `true` will ensure that all requests from the same ABAP session will end up in the same JCo session.

You can find an example implementation demonstrating stateful behavior with a simple counter in `examples/com/sap/conn/jco/examples/server/beginner/stateful`.

Server Security

Authentication

With the `JCoServerSecurityHandler`, you can check permissions of the user who initiated the RFC request. Besides the authorization also the authentication can be checked now. This can be done after the `JCoServerSecurityHandler` is registered on the `JCoServer` instance with `JCoServer.setSecurityHandler()` by implementing the method `checkAuthentication()`.



An example implementation is shown in the following code snippet. Depending on what kind of authentication data you want to evaluate, you can only check certain types. If authentication fails, a `JCoApplicationAuthenticationException` needs to be thrown.

```
class JCoServerSecurityHandlerImpl implements JCoServerSecurityHandler
{
    @Override
    public void checkAuthentication(JCoServerContextInfo serverCtxInfo,
                                   JCoServerAuthenticationData... authenticationData)
        throws JCoApplicationAuthenticationException
    {
        JCoServerAuthenticationData data;
        for (int i = 0; i < authenticationData.length; i++)
        {
            data = authenticationData[i];
            switch (data.getAuthenticationMode())
            {
                case SNC:
                    validateAuthenticationKey(data.getAuthenticationKey());
                    break;
                case SS0:
                    validateSS0Ticket(data.getSS0Ticket());
                    break;
                case X509:
                    validateX509Certificate(data.getX509Certificate());
                    break;
                default:
                    break;
            }
        }
    }
}
```

Further available access restrictions

Besides the `JCoServerSecurityHandler`, you can also allow access only to certain SAP systems or SNC names via server configuration. In order to achieve this, the property `jco.server.allowed_system_ids` or `jco.server.allowed_snc_partner_names` must be provided with a comma-separated list of system IDs or a pipe-separated list of SNC partner names.

High Availability Server

Classically, a `JCoServer` registers at one defined gateway. This new setup option lets you connect to multiple local gateways of an ABAP system simultaneously. The list of application servers for which a registration at their gateways should be done, is represented by a logon group and fetched from the message server. If no logon group is provided in the configuration, a registration at the gateways of all application servers of that ABAP system will be done. As a prerequisite in the destination configuration (transaction SM59), the gateway host and service fields must be empty, so that for each application server the local gateway is used when establishing the connection to the server. The following properties must be provided by the `ServerDataProvider` for such kind of server:

- `jco.server.mshost` – the message server host
- `jco.server.msserv` – the message server service name or port number, or

- `jco.server.system_id` – the system ID of the system the message server belongs to

Optionally, you can also define:

- `jco.server.group` – the logon group defined in ABAP which is identifying a set of application servers
- `jco.server.update_interval` – the update time how often the list of application servers is fetched from the message server

The set of properties can also be retrieved from the `JCoServer` instance with the respective getter methods.

IDOC SUPPORT FOR EXTERNAL JAVA APPLICATIONS

You can use SAP JCo 3.1 with the IDoc class library 3.1 that supports IDoc-based communication of external (non-SAP) Java applications with the AS ABAP.

Features

The Java IDoc class library provides the basic functionality for working with IDocs. This includes:

- Procuring and managing the metadata for IDoc types
- Navigation through an IDoc
- Constructing an IDoc
- Sending IDocs via the tRFC port in the ALE interface
- Receiving IDocs via the tRFC port in the ALE interface

An integrated IDoc XML processor enables the direct transformation from IDoc XML to the binary IDoc format and vice versa.



The IDoc XML processor is *not* an XML processor for *general* XML conversion purposes.

Implementation Considerations

The IDoc class library is a separate software component you can download in addition to SAP JCo 3.1 separately from SAP Service Marketplace.

In contrast to former versions, the current IDoc class library 3.1 is based – like SAP JCo 3.1 – on a destination model for the communication with partner systems.

Further Information

You can find detailed information on the IDoc class library in the Javadocs of the IDoc class library installation files.

An example for using the IDoc class library for IDoc communication via SAP JCo 3.1 (acting as a client or server) is included in `IDocClientExample.java` and `IDocServerExample.java`.