SAP Java Connector (Standalone Version)

HELP.BCMIDCONF

Release 3.0

Copyright

Icons in Body Text

Icon	Meaning
⚠	Caution
❖	Example
💡	Note
⬆	Recommendation
◇	Syntax

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see *Help on Help* → *General Information Classes and Information Classes for Business Information Warehouse* on the first page of any version of *SAP Library*.

Typographic Conventions

Type Style	Description
Example text	Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.
`Example text`	Emphasized words or phrases in body text, graphic titles, and table titles.
EXAMPLE TEXT	Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.
`EXAMPLE TEXT`	Keys on the keyboard, for example, `F2` or `ENTER`.

Document Version

	Version 8.1	**03-21-2013**

Contents

SAP Java Connector

Purpose

SAP Java Connector (SAP 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 implemented 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 and SAP Business Connector
- 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 exclusiveliy describes the standalone version of JCo 3.0 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 use the IDoc Class Library 3.0 additionally.
- You can access the SAP JCo and IDoc class library installation files at **`service.sap.com/connectors`**.

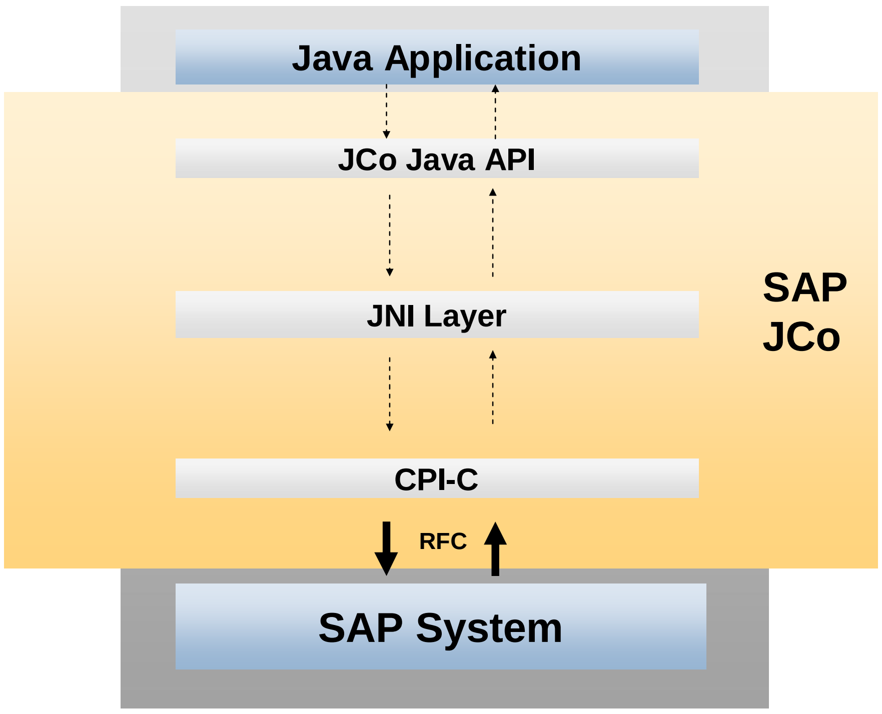
SAP JCo Functions

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

- SAP JCo is based on the *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 mySAP components that have *BAPI* or *RFMs* (*Remote Function Modules*).
- You can execute function calls *inbound* (Java client calls BAPI or RFM) and *outbound* (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 **`service.sap.com/connectors`**.

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 #1077727 for detailed information on supported platforms.

Procedure

The following instructions apply for Windows32 and Linux operating systems. The instructions for the installation of SAP JCo on other operating systems are included in the corresponding download files.

1. Create a directory, for example C:\SAPJCo, and extract the JCo .zip file into this directory.
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 SAP JCo .zip file are necessary:

- `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 download .zip file also contains the *docs* directory that contains the **Javadocs** for SAP JCo. The *Javadocs* contain an overview of all SAPJCo classes and interfaces, together with a detailed description of the corresponding objects, methods, parameters, and fields. Start with the file index.html (`<drive>:\<SAPJCo>\docs\jco\index.html`).

SAP JCo Customizing and Integration

After installiation 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 integration:

Interface	Use
`ClientPassportManager`	JDSR Passport Manager Interface for client connections to an SAP ABAP application server backend.
`DestinationDataProvider`	Provides the properties for a client connection to a remote SAP system.
`ServerDataProvider`	Provides the properties for a `JCoServer`.
`ServerPassportManager`	JDSR Passport Manager Interface for server connections to an SAP application server ABAP backend.
`SessionReferenceProvider`	Can be implemented by a runtime environment that has a session concept in order to provide JCo a simple reference to a session.

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 calls you have to implement `SessionReferenceProvider`. This interface connects JCo with session management.

You can find details and an example implementation in the JCo installation directory: *example/MultiThreadedExample.txt*.

Client Programming

The following section provides an overview of the main elements of client programming when using SAP JCo 3.0 as a standalone component (External (*non*-SAP) Java calls AS ABAP):

Topic/Activity	**Section**
Setting up connections	*Establishing a Connection with an AS ABAP*
Executing an RFM in an SAP System	*Executing Functions*
Access to tables and navigation in tables	*Access to Tables* *Processing Tables*
Executing a Function with different parameter values	*Setting Scalar Import Parameters*
Working with a multi threading environment	*Using Multi Threading*
Complete example program for JCo Client	*Example Program JCo Client*
Creating a JCo repository *JCoFunction* object	*SAP JCo Repository*
Assigning data types	*Mapping ABAP and Java Data Types*
Using suitable *getter* methods	*Type-Specific Getter Methods*
Generic processing of fields	*Interface JCoField*
Optimizing performance by deactivating non-essential BAPI parameters	*Deactivating Parameters*
Handling exceptions in SAP JCo	*Exception Handling*

More Information

For an overview of all SAP JCo classes, objects, methods, parameters, and fields, see the **Javadocs**. These HTML files are available in the *docs* directory of the SAPJCo installation.

Establishing a Connection to an AS ABAP

In JCo 3.0, the connection setup is no longer implemented *explicitly* using a single or pooled connection.

Instead, the type of connection is determined only by the connection properties (*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 point to the same ABAP system.

By specifying the destination name, the corresponding connection is set up.

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* ist an interface that should be implemented depending on the corresponding environment.

The implementation delivered by JCo distribution reads the destination configuration from the file system. However, this procedure is recommended only for development and testing. Within the 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 practice you should avoid this for security reasons.

Defining Destinations

```
public class StepByStepClient
{
    static String DESTINATION_NAME1 = "ABAP_AS_WITHOUT_POOL";
    static String DESTINATION_NAME2 = "ABAP_AS_WITH_POOL";
    static
    {
        Properties connectProperties = new Properties();

connectProperties.setProperty(DestinationDataProvider.JCO_ASHOST,
"ls4065");

connectProperties.setProperty(DestinationDataProvider.JCO_SYSNR,
"85");

connectProperties.setProperty(DestinationDataProvider.JCO_CLIENT,
"800");

connectProperties.setProperty(DestinationDataProvider.JCO_USER,
"homo faber");

connectProperties.setProperty(DestinationDataProvider.JCO_PASSWD,
"alaska");
connectProperties.setProperty(DestinationDataProvider.JCO_LANG,
"en");
        createDestinationDataFile(DESTINATION_NAME1,
connectProperties);


connectProperties.setProperty(DestinationDataProvider.JCO_POOL_CAPA
CITY, "3");

connectProperties.setProperty(DestinationDataProvider.JCO_PEAK_LIMI
T,    "10");
        createDestinationDataFile(DESTINATION_NAME2,
connectProperties);


    }


    static void createDestinationDataFile(String destinationName,
Properties connectProperties)
    {
        File destCfg = new File(destinationName+".jcoDestination");
        try
        {
            FileOutputStream fos = new FileOutputStream(destCfg,
false);
            connectProperties.store(fos, "for tests only !");
            fos.close();
        }
        catch (Exception e)
        {
            throw new RuntimeException("Unable to create the
destination files", e);
        }
    }
```

Setting up Connections

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

Because the connection type (direct or pool connection) is determined by the destination configuration, it is set implicitly by specifying the destination name.

Direct Connection to an AS ABAP

```
public static void step1Connect() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME1);
        System.out.println("Attributes:");
        System.out.println(destination.getAttributes());
        System.out.println();
    }
```

Pool Connection to an AS ABAP

```
public static void step2ConnectUsingPool() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        destination.ping();
        System.out.println("Attributes:");
        System.out.println(destination.getAttributes());
        System.out.println();
    }
```

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 need to declare a *stateful* connection explicitly.

Procedure

For a *stateful* connection, use the statements `JCoContext.begin(destination)`and `JCoContext.end(destination)`.

Example

JCo Client: Stateful Connection

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

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

Executing Functions

In the following example a function is executed by calling a function module and a structure is accessed.

Example

You want to call functions `STFC_CONNECTION` and `RFC_SYSTEM_INFO`.

1. You call a destination and the corresponding function.
2. All function parameters can be called using the methods *getImportParameterList()*, *getExportParameterList()*, and *getTableParameterList()*.
3. The method *getStructure()* facilitates access to any structure parameter in an import or export parameter list.

Executing Simple Functions

```
public static void step3SimpleCall() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function =
destination.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(destination);
        }
        catch(AbapException e)
        {
            System.out.println(e.toString());
            return;
        }

        System.out.println("STFC_CONNECTION finished:");
        System.out.println(" Echo: " +
function.getExportParameterList().getString("ECHOTEXT"));
        System.out.println(" Response: " +
function.getExportParameterList().getString("RESPTEXT"));
        System.out.println();
    } public static void step3SimpleCall() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function =
destination.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(destination);
        }
        catch(AbapException e)
        {
            System.out.println(e.toString());
            return;
        }

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

Accessing a Structure

```
 public static void step3WorkWithStructure() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function =
destination.getRepository().getFunction("RFC_SYSTEM_INFO");
        if(function == null)
            throw new RuntimeException("RFC_SYSTEM_INFO not found
in SAP.");

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

        JCoStructure exportStructure =
function.getExportParameterList().getStructure("RFCSI_EXPORT");
        System.out.println("System info for " +
destination.getAttributes().getSystemID() + ":\n");
        for(int i = 0; i <
exportStructure.getMetaData().getFieldCount(); i++)
        {

System.out.println(exportStructure.getMetaData().getName(i) + ":\t"
+ exportStructure.getString(i));
        }
        System.out.println();

        //JCo still supports the JCoFields, but direct access via
getXX is more efficient as field iterator
        System.out.println("The same using field iterator: \nSystem
info for " + destination.getAttributes().getSystemID() + ":\n");
        for(JCoField field : exportStructure)
        {
            System.out.println(field.getName() + ":\t" +
field.getString());
        }
        System.out.println();
    }
```

Starting a SAP GUI

When calling a function module in the SAP system it may be necessary to start a 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 a 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 a SAP GUI from your client program, proceed as follows:

Windows:

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

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

Possible values are:

0: no SAPGUI (default)

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

2: attach a visible SAPGUI.

Unix:

For Unix systems a Java SAP GUI is required.

Set the envrionment variable via:

`setenv SAPGUI <pfad zum SAPGUI startskript> (tcsh)` oder

```
set SAPGUI=<pfad zum SAPGUI startskript> (bash)
```

For Mac OS the path name could look like this:

```
"/Applications/SAP Clients/SAPGUI 7.10rev7.3/SAPGUI
7.10rev7.3.app/Contents/MacOS/SAPGUI"
```

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

```
/opt/SAPClients/SAPGUI7.10rev7.3/bin/sapgui
```

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

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

Access to Tables

Activities

In the next step you call the *CompanyCode.GetList* BAPI and a table of all the *company codes* is displayed. The corresponding RFM is called BAPI_COMPANYCODE_GETLIST. This RFM does not contain any import parameters.

1. First, get the table by accessing the table parameter list (*getTableParameterList()*).
2. Within this list, access the actual table (*getTable()*). The interface *JCoTable* contains all methods that are available for *JCoStructure*, together with additional methods for navigation in a table. A table can have any number of rows, or can also have no rows. The diagram below shows navigation with the method *setRow()*, 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 *setRow()*, you can also use the method *nextRow()*, as displayed in the lower section of the diagram.

Accessing a Table

```
public static void step4WorkWithTable() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function =
destination.getRepository().getFunction("BAPI_COMPANYCODE_GETLIST")
;
        if(function == null)
            throw new RuntimeException("BAPI_COMPANYCODE_GETLIST
not found in SAP.");

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

        JCoStructure returnStructure =
function.getExportParameterList().getStructure("RETURN");
        if (!
(returnStructure.getString("TYPE").equals("")||returnStructure.getS
tring("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"));
        }
```

Setting Scalar Import Parameters

This step calls the BAPI *CompanyCode.GetDetail* for each *company code*. The corresponding RFM is `BAPI_COMPANYCODE_GETDETAIL`. For this RFM, you need to set the scalar import parameter `COMPANYCODEID`.

Activities

1. To access the import parameter list, use *getImportParameterList()*.
2. The value of the scalar parameter is set using *setValue()*, in which first the value, and then the name are entered. There are many different versions of *setValue()* in the SAP JCo to support all existing data types.
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()* is also available in *JCoStructure*, *JCoTable*, and *JCoField* . You can therefore set values from structure fields and fields in a row of a table.

Setting Scalar Import Parameters

```
codes.firstRow();
        for (int i = 0; i < codes.getNumRows(); i++, 
codes.nextRow())
        {
            function = 
destination.getRepository().getFunction("BAPI_COMPANYCODE_GETDETAIL
");
            if (function == null)
                throw new 
RuntimeException("BAPI_COMPANYCODE_GETDETAIL not found in SAP.");


function.getImportParameterList().setValue("COMPANYCODEID", 
codes.getString("COMP_CODE"));

function.getExportParameterList().setActive("COMPANYCODE_ADDRESS",f
alse);

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

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

            JCoStructure detail = 
function.getExportParameterList().getStructure("COMPANYCODE_DETAIL"
);

            System.out.println(detail.getString("COMP_CODE") + '\t' 
+
                               detail.getString("COUNTRY") + '\t' +
                               detail.getString("CITY"));
        }//for
    }
```

4. You call *firstRow()* before the loop, because the row pointer for the table is located on the last row as a result of the previous loop.

Note that in the BAPI error handling displayed here, “W“ (warning) is accepted as well as *empty string* or ”S“ (success).

This occurs because this particular BAPI sometimes outputs the warning that address data has not been maintained (in the structure parameter

COMPANYCODE_ADDRESS). In the current example program this parameter is not relevant, so the warning can be ignored. In a productive program, error handling must be more precisely configured and must also check the number of the warning message.

Using Multi-Threading

Most applications run in several threads. JCo classes like repository or desination 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), internal JCo session management 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 has to provide an adequate implementation of `SessionReferenceProvider`. This interface enables the JCo runtime to assign the processes in different threads to a unique session.

If you want to work with multi-threading in your JCo Client, you may need the `SessionReferenceProvider` interface. You can create a simple session reference using this interface in the JCo. The session reference assigns calls from different threads to a unique session ID.

In a multi-thread environment, the use of container objects (for example, *JCoTable* objects) from different threads must be implemented carefully. Note that it is not possible to make multiple concurrent SAP calls for the same direct connection.

The following example specifies a `MultiStepJob` with several execution steps. After each execution step a change of threads will be performed.

The program starts a defined number of threads using two different jobs, one for stateless and the other for stateful communication: `StatelessMultiStepExample` and `StatefulMultiStepExample`. Both jobs invoke the same RFC function module. However, `StatefulMultiStepExample` uses `JCoContext.begin und JCoContext.end to` establish a stateful connection (see section Setting Up Stateful Connections).

The example program uses the implementation `MySessionReferenceProvider` of the interface `SessionReferenceProvider` to perform stateful calls across several steps.

Example

Multi Threading Scenario

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Properties;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import com.sap.conn.jco.JCoContext;
import com.sap.conn.jco.JCoDestination;
import com.sap.conn.jco.JCoDestinationManager;
import com.sap.conn.jco.JCoException;
import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.JCoFunctionTemplate;
import com.sap.conn.jco.ext.DestinationDataProvider;
import com.sap.conn.jco.ext.Environment;
import com.sap.conn.jco.ext.JCoSessionReference;
import com.sap.conn.jco.ext.SessionException;
import com.sap.conn.jco.ext.SessionReferenceProvider;

public class MultiThreadedExample
{
    static String DESTINATION_NAME1 = "ABAP_AS_WITHOUT_POOL";
    static String DESTINATION_NAME2 = "ABAP_AS_WITH_POOL";
    static
    {
        Properties connectProperties = new Properties();

connectProperties.setProperty(DestinationDataProvider.JCO_ASHOST,
"binmain");

connectProperties.setProperty(DestinationDataProvider.JCO_SYSNR,
"53");

connectProperties.setProperty(DestinationDataProvider.JCO_CLIENT,
"000");

connectProperties.setProperty(DestinationDataProvider.JCO_USER,
"JCOTEST");

connectProperties.setProperty(DestinationDataProvider.JCO_PASSWD,
"JCOTEST");

connectProperties.setProperty(DestinationDataProvider.JCO_LANG,
"en");
        createDataFile(DESTINATION_NAME1, "jcoDestination",
connectProperties);


connectProperties.setProperty(DestinationDataProvider.JCO_POOL_CA
PACITY, "3");

connectProperties.setProperty(DestinationDataProvider.JCO_PEAK_LI
MIT,    "10");
        createDataFile(DESTINATION_NAME2, "jcoDestination",
connectProperties);
    }

    static void createDataFile(String name, String suffix,
Properties properties)
    {
        File cfg = new File(name+"."+suffix);
        if(!cfg.exists())
        {
            try
            {
                FileOutputStream fos = new FileOutputStream(cfg,
false);
                properties.store(fos, "for tests only !");
                fos.close();
            }
            catch (Exception e)
            {
                throw new RuntimeException("Unable to create the
destination file " + cfg.getName(), e);
            }
        }
    }

    static void createDestinationDataFile(String destinationName,
Properties connectProperties)
    {
        File destCfg = new
File(destinationName+".jcoDestination");
        try
        {
            FileOutputStream fos = new FileOutputStream(destCfg,
false);
            connectProperties.store(fos, "for tests only !");
            fos.close();
        }
        catch (Exception e)
        {
            throw new RuntimeException("Unable to create the
destination files", e);
        }
    }

    interface MultiStepJob
    {
        boolean isFinished();
        public void runNextStep();
        String getName();
        public void cleanUp();
    }

    static class StatelessMultiStepExample implements
MultiStepJob
    {
        static AtomicInteger JOB_COUNT = new AtomicInteger(0);
        int jobID = JOB_COUNT.addAndGet(1);
        int calls;
        JCoDestination destination;

        int executedCalls = 0;
        Exception ex = null;
        int remoteCounter;

        StatelessMultiStepExample(JCoDestination destination, int
calls)
        {
            this.calls = calls;
            this.destination = destination;
        }

        public boolean isFinished() { return executedCalls ==
calls || ex != null; }
        public String getName() { return "stateless Job-"+jobID;
}

        public void runNextStep()
        {
            try
            {
                JCoFunction incrementCounter = 
incrementCounterTemplate.getFunction();
                incrementCounter.execute(destination);
                JCoFunction getCounter = 
getCounterTemplate.getFunction();
                executedCalls++;

                if(isFinished())
                {
                    getCounter.execute(destination);
                    remoteCounter = 
getCounter.getExportParameterList().getInt("GET_VALUE");
                }
            }
            catch(JCoException je)
            {
                ex = je;
            }
            catch(RuntimeException re)
            {
                ex = re;
            }
        }

        public void cleanUp()
        {
            StringBuilder sb = new StringBuilder("Task 
").append(getName()).append(" is finished ");
            if(ex!=null)
                sb.append("with exception 
").append(ex.toString());
            else
                sb.append("successful. Counter is 
").append(remoteCounter);
            System.out.println(sb.toString());
        }

    }

    static class StatefulMultiStepExample extends
```

```
StatelessMultiStepExample
    {
        StatefulMultiStepExample(JCoDestination destination, int 
calls)
        {
            super(destination, calls);

        }

        @Override
        public String getName() { return "stateful Job-"+jobID; }

        @Override
        public void runNextStep()
        {
            if(executedCalls == 0)
                JCoContext.begin(destination);
            super.runNextStep();
        }

        @Override
        public void cleanUp()
        {
            try
            {
                JCoContext.end(destination);
            }
            catch (JCoException je)
            {
                ex = je;
            }
            super.cleanUp();
        }
    }

    static class MySessionReferenceProvider implements 
SessionReferenceProvider
    {
        public JCoSessionReference 
getCurrentSessionReference(String scopeType)
        {
            MySessionReference sesRef = 
WorkerThread.localSessionReference.get();
            if(sesRef != null)
                return sesRef;

            throw new RuntimeException("Unknown thread:" + 
Thread.currentThread().getId());
        }

        public boolean isSessionAlive(String sessionId)
        {
            Collection<MySessionReference> availableSessions = 
WorkerThread.sessions.values();
            for(MySessionReference ref : availableSessions)
            {
                if(ref.getID().equals(sessionId))
                    return true;
            }
            return false;
        }

        public void jcoServerSessionContinued(String sessionID) 
throws SessionException
        {
        }

        public void jcoServerSessionFinished(String sessionID)
        {
        }

        public void jcoServerSessionPassivated(String sessionID) 
throws SessionException
        {
        }

        public JCoSessionReference jcoServerSessionStarted() 
throws SessionException
        {
            return null;
        }
    }
    static class MySessionReference implements
JCoSessionReference
    {
        static AtomicInteger atomicInt = new AtomicInteger(0);
        private String id = "session-
"+String.valueOf(atomicInt.addAndGet(1));;

        public void contextFinished()
        {
        }

        public void contextStarted()
        {
        }

        public String getID()
        {
            return id;
        }

    }

    static class WorkerThread extends Thread
    {
        static Hashtable<MultiStepJob, MySessionReference>
sessions = new Hashtable<MultiStepJob, MySessionReference>();
        static ThreadLocal<MySessionReference>
localSessionReference = new ThreadLocal<MySessionReference>();

        private CountDownLatch doneSignal;
        WorkerThread(CountDownLatch doneSignal)
        {
            this.doneSignal = doneSignal;
        }

        @Override
        public void run()
        {
            try
            {
                for(;;)
                {
                    MultiStepJob job = queue.poll(10, 
TimeUnit.SECONDS);

                    //stop if nothing to do
                    if(job == null)
                        return;

                    MySessionReference sesRef = 
sessions.get(job);
                    if(sesRef == null)
                    {
                        sesRef = new MySessionReference();
                        sessions.put(job, sesRef);
                    }
                    localSessionReference.set(sesRef);

                    System.out.println("Task "+job.getName()+" is 
started.");
                    try
                    {
                        job.runNextStep();
                    }
                    catch (Throwable th)
                    {
                        th.printStackTrace();
                    }

                    if(job.isFinished())
                    {
                        System.out.println("Task 
"+job.getName()+" is finished.");
                        sessions.remove(job);
                        job.cleanUp();
                    }
                    else
                    {
                        System.out.println("Task 
"+job.getName()+" is passivated.");
                        queue.add(job);
                    }
                    localSessionReference.set(null);
                }
            }
            catch (InterruptedException e)
            {
                //just leave
            }
            finally
            {
                doneSignal.countDown();
            }
        }
    }

    private static BlockingQueue<MultiStepJob> queue = new
LinkedBlockingQueue<MultiStepJob>();
    private static JCoFunctionTemplate incrementCounterTemplate,
getCounterTemplate;


    static void runJobs(JCoDestination destination, int jobCount,
int threadCount)
    {
        System.out.println(">>> Start");
        for(int i = 0; i < jobCount; i++)
        {
            queue.add(new StatelessMultiStepExample(destination,
10));
            queue.add(new StatefulMultiStepExample(destination,
10));
        }

        CountDownLatch doneSignal = new
CountDownLatch(threadCount);
        for(int i = 0; i < threadCount; i++)
            new WorkerThread(doneSignal).start();

        System.out.print(">>> Wait ... ");
        try
        {
            doneSignal.await();
        }
        catch (InterruptedException ie)
```

```
            {
                //just leave
            }
            System.out.println(">>> Done");
        }

        public static void main(String[] argv)
        {
            //JCo.setTrace(5, ".");
            Environment.registerSessionReferenceProvider(new
MySessionReferenceProvider());
            try
            {
                JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
                incrementCounterTemplate =
destination.getRepository().getFunctionTemplate("Z_INCREMENT_COUN
TER");
                getCounterTemplate =
destination.getRepository().getFunctionTemplate("Z_GET_COUNTER");
                if(incrementCounterTemplate == null ||
getCounterTemplate == null)
                    throw new RuntimeException("This example cannot
run without Z_INCREMENT_COUNTER and Z_GET_COUNTER functions");

                runJobs(destination, 5, 2);
            }
            catch(JCoException je)
            {
                je.printStackTrace();
            }

        }
    }
```

Example Program JCo Client

In this section you find an example of a complete JCo client program. It is made up of the code examples from the activities described in the previous sections:

- Defining Destinations
- Setting Up Connections
- Executing a Function with Access to a Structure

- Accessing Tables
- Setting Scalar Import Parameters

Example

JCo Client

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.List;
import java.util.Properties;

import com.sap.conn.jco.AbapException;
import com.sap.conn.jco.JCoDestination;
import com.sap.conn.jco.JCoDestinationManager;
import com.sap.conn.jco.JCoException;
import com.sap.conn.jco.JCoField;
import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.JCoStructure;
import com.sap.conn.jco.JCoTable;
import com.sap.conn.jco.ext.DestinationDataProvider;
import com.sap.conn.jco.ext.JCoSessionReference;
import com.sap.conn.jco.ext.SessionException;
import com.sap.conn.jco.ext.SessionReferenceProvider;

public class StepByStepClient
{
    static String DESTINATION_NAME1 = "ABAP_AS_WITHOUT_POOL";
    static String DESTINATION_NAME2 = "ABAP_AS_WITH_POOL";
    static
    {
        Properties connectProperties = new Properties();

connectProperties.setProperty(DestinationDataProvider.JCO_ASHOST,
"ls4065");

connectProperties.setProperty(DestinationDataProvider.JCO_SYSNR,
"85");

connectProperties.setProperty(DestinationDataProvider.JCO_CLIENT,
"800");
```

```
connectProperties.setProperty(DestinationDataProvider.JCO_USER,
"homo faber");

connectProperties.setProperty(DestinationDataProvider.JCO_PASSWD,
"alaska");

connectProperties.setProperty(DestinationDataProvider.JCO_LANG,
"en");
        createDestinationDataFile(DESTINATION_NAME1,
connectProperties);


connectProperties.setProperty(DestinationDataProvider.JCO_POOL_CAPA
CITY, "3");

connectProperties.setProperty(DestinationDataProvider.JCO_PEAK_LIMI
T,    "10");
        createDestinationDataFile(DESTINATION_NAME2,
connectProperties);

    }

    static void createDestinationDataFile(String destinationName,
Properties connectProperties)
    {
        File destCfg = new File(destinationName+".jcoDestination");
        try
        {
            FileOutputStream fos = new FileOutputStream(destCfg,
false);
            connectProperties.store(fos, "for tests only !");
            fos.close();
        }
        catch (Exception e)
        {
            throw new RuntimeException("Unable to create the
destination files", e);
        }
    }

    public static void step1Connect() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME1);
```

```
        System.out.println("Attributes:");
        System.out.println(destination.getAttributes());
        System.out.println();
    }

    public static void step2ConnectUsingPool() throws JCoException
    {
        JCoDestination destination = 
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        destination.ping();
        System.out.println("Attributes:");
        System.out.println(destination.getAttributes());
        System.out.println();
    }

    public static void step3SimpleCall() throws JCoException
    {
        JCoDestination destination = 
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function = 
destination.getRepository().getFunction("STFC_CONNECTION");
        If (function == null)
            throw new RuntimeException("BAPI_COMPANYCODE_GETLIST 
not found in SAP.");

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

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

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

    public static void step3WorkWithStructure() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function =
destination.getRepository().getFunction("RFC_SYSTEM_INFO");
        if (function == null)
            throw new RuntimeException("BAPI_COMPANYCODE_GETLIST
not found in SAP.");

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

        JCoStructure exportStructure =
function.getExportParameterList().getStructure("RFCSI_EXPORT");
        System.out.println("System info for " +
destination.getAttributes().getSystemID() + ":\n");
        for(int i = 0; i <
exportStructure.getMetaData().getFieldCount(); i++)
        {

System.out.println(exportStructure.getMetaData().getName(i) + ":\t"
+ exportStructure.getString(i));
        }
        System.out.println();

        //JCo still supports the JCoFields, but direct access via
getXX is more efficient as field iterator
        System.out.println("The same using field iterator: \nSystem
info for " + destination.getAttributes().getSystemID() + ":\n");
        for(JCoField field : exportStructure)
        {
            System.out.println(field.getName() + ":\t" +
field.getString());
        }
        System.out.println();
    }

    public static void step4WorkWithTable() throws JCoException
    {
        JCoDestination destination =
JCoDestinationManager.getDestination(DESTINATION_NAME2);
        JCoFunction function =
destination.getRepository().getFunction("BAPI_COMPANYCODE_GETLIST")
;
        if (function == null)
            throw new RuntimeException("BAPI_COMPANYCODE_GETLIST
not found in SAP.");

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

        JCoStructure returnStructure =
function.getExportParameterList().getStructure("RETURN");
        if (!
(returnStructure.getString("TYPE").equals("")||returnStructure.getS
tring("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"));
```

```
        }

        codes.firstRow();
        for (int i = 0; i < codes.getNumRows(); i++, 
codes.nextRow())
        {
            function = 
destination.getRepository().getFunction("BAPI_COMPANYCODE_GETDETAIL
");
            if (function == null)
                throw new 
RuntimeException("BAPI_COMPANYCODE_GETDETAIL not found in SAP.");


function.getImportParameterList().setValue("COMPANYCODEID", 
codes.getString("COMP_CODE"));

function.getExportParameterList().setActive("COMPANYCODE_ADDRESS",f
alse);

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

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

            JCoStructure detail = 
function.getExportParameterList().getStructure("COMPANYCODE_DETAIL"
);
```

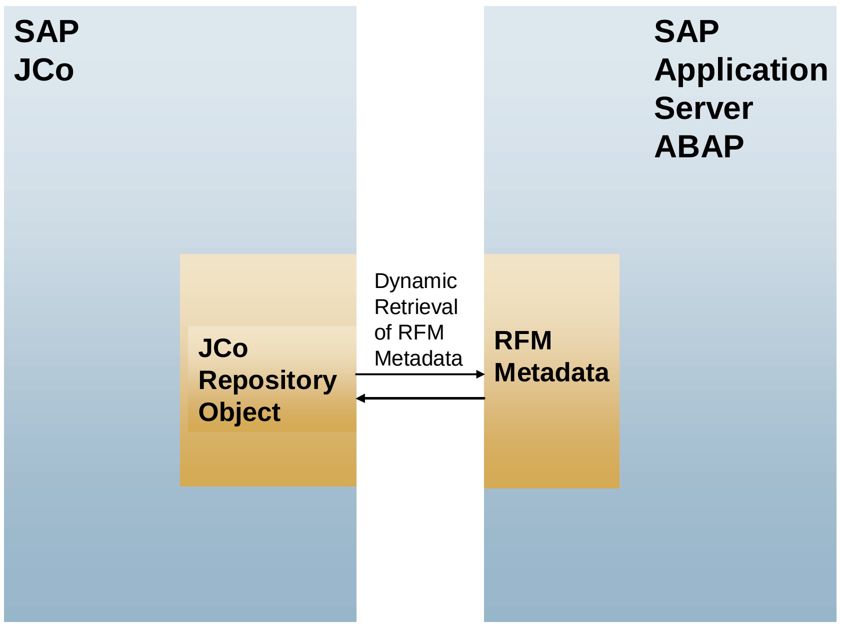
```
            System.out.println(detail.getString("COMP_CODE") + '\t'
+
                                   detail.getString("COUNTRY") + '\t' +
                                   detail.getString("CITY"));
        }
  }
```

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) 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.

SAP JCo Repository


SAP JCo
SAP Application Server ABAP
JCo Repository Object
Dynamic Retrieval of RFM Metadata
RFM Metadata

Obtaining a Repository

Activities

- For the JCo Repository, you need the following interfaces:
  - **JCoRepository**: Contains the runtime metadata of the RFMs.
  - **IFunctionTemplate**: Contains the metadata for an RFM.
  - **JCoFunction**: Represents an RFM with all its corresponding parameters.
- Make sure that the user ID for the Repository has all the required authorizations for accessing the metadata of the AS ABAP.

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

  You have to lock the *change* function of metadata before using them to avoid any unintended changes.

Examples

Interfaces

```
JCoRepository

JCoFunctionTemplate

JCoFunction

JCoParameterList

JCoStructure

JCoTable
```

Creating a JCo Repository

```
JCoRepository mRepository;

mRepository = destination.getRepository ();
```

Creating JCoFunction Objects

Activities

To create a `JCoFunction` object, proceed as follows:

1. Execute the method `getFunction()` against the repository.
2. Execute the method `getFunctionTemplate()` against the repository.
3. Execute the method `getFunction()` against the template.

In addition to containing 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 similar to 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.

Mapping of ABAP and Java Data Types

A data structure is made up of individual fields and each field is assigned to a particular data type. Because ABAP uses different data types to 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.*

Data Type Mapping

ABAP Type	Description	Data Type	Java Type Code
C	Character	String	JCoMetadata.TYPE_CHAR
N	Numerical Character	String	JCoMetadata.TYPE_NUM
X	Binary Data	Byte ()	JCoMetadata.TYPE_BYTE
P	Binary Coded Decimal	Big Decimal	JCoMetadata.TYPE_BCD
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
`decfloat16`	Decimal floating point 8 bytes (IEEE 754r)	BigDecimal	JCoMetadata.TYPE_DECF16
`decfloat34`	Decimal floating point 16 bytes (IEEE 754r)	BigDecimal	JCoMetadata.TYPE_DECF34
g	String (variable length)	String	JCoMetadata.TYPE_STRING
y	Raw String (variable length)	Byte ()	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-byte string with the format HHMMSS
- ABAP data type D is an 8-byte 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 need to distinguish between whether a particular field contains an ABAP date value or an ABAP time value.

Type-Specific 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.

Type-Specific Getter Methods

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()
JCoMetadata.TYPE_STRING	String getString()
JCoMetadata.TYPE_XSTRING	byte[ ] getByteArray()
JCoMetadata.TYPE_DECF16	BigDecimal getBigDecimal()
JCoMetadata.TYPE_DECF34	BigDecimal getBigDecimal()

Table Manipulation

In many applications it is not sufficient 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). This example uses the table returned by the BAPI *CompanyCode.GetList*.

Table Manipulation

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

- If the method *deleteRow()* is called without parameters, it deletes the current row. If you define a row number, the corresponding row is deleted.
- The method *appendRow()* adds a row at the end of the table. If you want to append multiple rows simultaneously, you can specify an *integer argument*. This leads to better performance than with adding rows individually.
- The method *insertRow(int)* inserts a row at any position in the table.
- The method *deleteAllRows()* deletes all rows of a table.

The following table summarizes the *JCoTable* methods that are not contained in *JCoStructur*e.

JCoTable Methods

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.

Interface JCoField

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. *JCoRecord* classes like *JCoStructure* or *JCoParameterList* offer *getFieldIterator()* as well as *iterator()* methods.

With the iterator you can access the single fields.

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*().

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, You are familiar with the *JCoField* interface. 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. There is no way to prevent the SAP system from returning a parameter, but you can prevent the SAP JCo forwarding a parameter to the Java layer. To do this, you simply need to declare the parameter as *inactive*, as displayed in the diagram below. This is particularly effective for larger tables that are returned by the SAP system.

Deactivating Parameters

```
function.getExportParameterList().


    setActive(false, "COMPANYCODE_ADDRESS");
```

Exception Handling

For exception handling, you need the following classes:

- **JCoException**: Basis class for exceptions.
  - o **ABAPException**: class for errors displayed in the RFM.

    `JCoAbapException` occurs if the ABAP code that you have called throws an exception.

    ABAPClassException: `AbapClassException` occurs if the ABAP code that you have called throws a class based exception.

    Class based exceptions are supported in ABAP with SAP NetWeaver release 740 or higher.
- **JcoRuntimeException**: Basis class for runtime exceptions.
  - o **JCoConversionException**: Special class for conversion errors.

    `ConversionException` is always thrown if you call a getter or setter method that requests a conversion, and this conversion fails.
  - o **XMLParserException**: Used with errors in the XML parser.

    `XMLParserException` is displayed by the XML parser if errors occur in RFC communication when using complex parameters.

SAP JCo throws exceptions as a subclass of `JCoException` or `JCoRuntimeException`.

An exception is normally thrown as a 'checked' exception, that is as an subclass of `JCoException`.

However, when using these 'normal' exceptions you must always decide whether an exception is to be handled explicitly or if it is to be added to the *throws clause*.

It is therefore recommended that you use *try/catch* in the code.

Exception Handling – Option 1

```
 public void executeFunction(JCO.Function function, String where)
    throws JCoException
{
    JCoDestination
destination=JCoDestinationManager.getDestination(where);
    function.execute(destination);
}
```

Exception Handling – Option 2

```
public int executeFunction(JCO.Function function, String where)
{
    int returnCode=0;
    try
    {
        JCoDestination
destination=JCoDestinationManager.getDestination(where);
        function.execute(destination);
    }
    catch (AbapException ex) //clause 1
    {
        handleAbapException(function, ex);
        returnCode=1;
    }
    catch (JCoException ex) //clause 2
    {
        logSystemFailure(ex);
        returnCode=2;
    }
    catch (JCoRuntimeException ex) //clause 3
    {
    }
    catch (Exception ex) //clause 4
    {
    }
    return returnCode;
}
```

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. If this is `NOT_FOUND`, a specific text is output, for all other exceptions, use `getMessage()` to generate a suitable text. 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-relevant problems. These include conversion errors and other errors that occur in SAP JCo.
- The third 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 can also be necessary 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 *docs*.

Server Programming

The following section explains how you can write your own JCo programs for servers if you use the standalone version of the SAP JCo.

A JCo server program implements functions that are called up by an ABAP Backend. The JCo server program is registered via the SAP Gateway and waits for inbound RFC calls.

- An RFC server program registers itself under a program ID to an SAP gateway (not for a specific SAP system).
- 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), the connection takes place with the corresponding JCo server program.
- Once an RFC function has been executed, the JCo Server waits for further RFC calls from the same or other SAP systems.
- 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.
- Using transaction SM59, you have defined a destination with connection type T (TCP/IP connection) in the SAP system.
- You have chosen the registration mode ("Registered server program" option under the "Technical settings" tab page) for this destination.
- The destination contains the required information about the SAP gateway and the registered RFC server program.

In the following sections you can find examples for programming important JCo server functions with the JCo API 3.0:

- Inbound RFC Connection (from AS ABAP)
- Java Programm for Establishing a Server Connection
- Implementing an Exception Listener
- Monitoring Server Connections
- Processing an ABAP Call
- Example Program JCo Server

Inbound RFC Connection (from AS ABAP)

This section provides an example of how you can establish a server-side RFC connection that originates from the SAP system.

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

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

ABAP Program for RFC Inbound Connection

```
DATA: REQUTEXT LIKE SY-LISEL,
      RESPTEXT LIKE SY-LISEL,
      ECHOTEXT LIKE SY-LISEL.

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

REQUTEXT = 'HELLO WORLD'.
RFCDEST = 'JCOSERVER01'. "corresponds to the destination name
defined in the SM59

CALL FUNCTION 'STFC_CONNECTION'
   DESTINATION RFCDEST
   EXPORTING
     REQUTEXT = REQUTEXT
   IMPORTING
     RESPTEXT = RESPTEXT
     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 Establishing a Server Connection

In the next step, you can write a Java program that establishes a server connection to a SAP gateway.

Procedure

To do this, you need to:

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

Example

Definition of Server Properties

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.Hashtable;
import java.util.Map;
import java.util.Properties;

import com.sap.conn.jco.JCoException;
import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.ext.DestinationDataProvider;
import com.sap.conn.jco.ext.ServerDataProvider;
import com.sap.conn.jco.server.DefaultServerHandlerFactory;
import com.sap.conn.jco.server.JCoServer;
import com.sap.conn.jco.server.JCoServerContext;
import com.sap.conn.jco.server.JCoServerErrorListener;
import com.sap.conn.jco.server.JCoServerExceptionListener;
import com.sap.conn.jco.server.JCoServerFactory;
import com.sap.conn.jco.server.JCoServerFunctionHandler;
import com.sap.conn.jco.server.JCoServerState;
import com.sap.conn.jco.server.JCoServerStateChangedListener;
import com.sap.conn.jco.server.JCoServerTIDHandler;


public class StepByStepServer
{
    static String SERVER_NAME1 = "SERVER";
    static String DESTINATION_NAME1 = "ABAP_AS_WITHOUT_POOL";
    static String DESTINATION_NAME2 = "ABAP_AS_WITH_POOL";
    static
    {
        Properties connectProperties = new Properties();

connectProperties.setProperty(DestinationDataProvider.JCO_ASHOST,
"ls4065");

connectProperties.setProperty(DestinationDataProvider.JCO_SYSNR,
"85");

connectProperties.setProperty(DestinationDataProvider.JCO_CLIENT,
"800");

connectProperties.setProperty(DestinationDataProvider.JCO_USER,
"homo faber");

connectProperties.setProperty(DestinationDataProvider.JCO_PASSWD,
"alaska");

connectProperties.setProperty(DestinationDataProvider.JCO_LANG,
"en");
        createDataFile(DESTINATION_NAME1, "jcoDestination",
connectProperties);



connectProperties.setProperty(DestinationDataProvider.JCO_POOL_CAPA
CITY, "3");

connectProperties.setProperty(DestinationDataProvider.JCO_PEAK_LIMI
T,    "10");
        createDataFile(DESTINATION_NAME2, "jcoDestination",
connectProperties);


        Properties servertProperties = new Properties();

servertProperties.setProperty(ServerDataProvider.JCO_GWHOST,
"binmain");

servertProperties.setProperty(ServerDataProvider.JCO_GWSERV,
"sapgw53");

servertProperties.setProperty(ServerDataProvider.JCO_PROGID,
"JCO_SERVER");

servertProperties.setProperty(ServerDataProvider.JCO_REP_DEST,
"ABAP_AS_WITH_POOL");

servertProperties.setProperty(ServerDataProvider.JCO_CONNECTION_COU
NT, "2");
        createDataFile(SERVER_NAME1, "jcoServer",
servertProperties);
    }


    static void createDataFile(String name, String suffix,
Properties properties)
    {
        File cfg = new File(name+"."+suffix);
        if(!cfg.exists())
        {
            try
            {
                FileOutputStream fos = new FileOutputStream(cfg,
false);
                properties.store(fos, "for tests only !");
                fos.close();
            }
            catch (Exception e)
            {
                throw new RuntimeException("Unable to create the
destination file " + cfg.getName(), e);
            }
        }
    }
```

JCo Server

```
       static class StfcConnectionHandler implements
JCoServerFunctionHandler
    {
        public void handleRequest(JCoServerContext serverCtx,
JCoFunction function)
        {
            System.out.println("----------------------------------
-----------------------------");
            System.out.println("call              : " +
function.getName());
            System.out.println("ConnectionId      : " +
serverCtx.getConnectionID());
            System.out.println("SessionId         : " +
serverCtx.getSessionID());
            System.out.println("TID               : " +
serverCtx.getTID());
            System.out.println("repository name   : " +
serverCtx.getRepository().getName());
            System.out.println("is in transaction : " +
serverCtx.isInTransaction());
            System.out.println("is stateful       : " +
serverCtx.isStatefulSession());
            System.out.println("----------------------------------
-----------------------------");
            System.out.println("gwhost: " +
serverCtx.getServer().getGatewayHost());
            System.out.println("gwserv: " +
serverCtx.getServer().getGatewayService());
            System.out.println("progid: " +
serverCtx.getServer().getProgramID());
            System.out.println("------------------------------------
------------------------------");
            System.out.println("attributes  : ");

System.out.println(serverCtx.getConnectionAttributes().toString());
            System.out.println("------------------------------------
------------------------------");
            System.out.println("req text: " +
function.getImportParameterList().getString("REQUTEXT"));
            function.getExportParameterList().setValue("ECHOTEXT",
function.getImportParameterList().getString("REQUTEXT"));
            function.getExportParameterList().setValue("RESPTEXT",
"Hello World");
        }
    }
    static void step1SimpleServer()
    {
        JCoServer server;
        try
        {
            server = JCoServerFactory.getServer(SERVER_NAME1);
        }
        catch(JCoException ex)
        {
            throw new RuntimeException("Unable to create the server
" + SERVER_NAME1 + ", because of " + ex.getMessage(), ex);
        }

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

        server.start();
        System.out.println("The program can be stopped using
<ctrl>+<c>");
    }
```

Implementing an Exception Listener

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

The application must process the exceptions separately in the method `handleRequest()`(that is, the exceptions that it generates itself). Exceptions from the application coding are not passed on to the Listener.

To define this listener, create a class that implements `JCoServerExceptionListener` and `JCoServerStateChangedListener`:

Exception and Error Listener

```
static class MyThrowableListener implements JCoServerErrorListener,
JCoServerExceptionListener
    {

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

        public void serverExceptionOccurred(JCoServer jcoServer,
String connectionId, JCoServerContextInfo serverCtx, Exception
error)
        {
            System.out.println(">>> Error occured 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 the server connection:

Monitoring Server Connections

```
  static class MyStateChangedListener implements
JCoServerStateChangedListener
    {
        public void serverStateChangeOccurred(JCoServer server,
JCoServerState oldState, JCoServerState newState)
        {

            // Defined states are: STARTED, DEAD, ALIVE, STOPPED;
            // see JCoServerState class for details.
            // 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 that processes the call must trigger its execution may only allow ABAP exceptions.

All other exceptions that are thrown by `handleRequest` are treated as system *failures*.

Processing an ABAP Call

```
 public void handleRequest(JCoServerContext serverCtx, JCoFunction
function)
    {
        try
        {
            //process the request
        }
        catch(Exception e)
        {
            throw new AbapException("a group of the exception",
"text");
        }
    }
```

The exception and error listeners are not informed about the exception case. This takes place in `handleRequest`.

Processing a tRFC/qRFC /bgRFC Call

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

If the following ABAP statement is called up:

CALL FUNCTION 'STFC_CONNECTION'

DESTINATION RFCDEST

IN BACKGROUND TASK (*bgRFC*: IN BACKGROUND *UNIT*)

fRFC/qRFC/bgRFC processing takes place.

The following section describes sample implementations for the use of tRFC/qRFC and bgRFC:

tRFC/qRFC Calls

For the use of tRFC, 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.

2. Calls the function module.
3. `commit(String tid)` or `rollback(String tid)`
4. Depending on the result of the function distribution. If `handleRequest` has not thrown any exceptions, `onCommit` is called.
5. *protected void* method `onConfirmTID(String tid)`.

At this point, the application is informed that execution has been ended by the call with the specified 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.

Processing a tRFC/qRFC **Call**

```
static class MyTIDHandler implements JCoServerTIDHandler
    {

        Map<String, TIDState> availableTIDs = new Hashtable<String,
TIDState>();

        public boolean checkTID(JCoServerContext serverCtx, String
tid)
        {
        // This example uses a hash table to store status
        //information.
        // Usually you would use a database. If the DB is down,
        // throw a RuntimeException at this point. JCo will then
        // abort the tRFC and the SAP backend will try again
        // later.

           System.out.println("TID Handler: checkTID for " + tid);
            TIDState state = availableTIDs.get(tid);
            if(state == null)
            {
                availableTIDs.put(tid, TIDState.CREATED);
                return true;
            }

            if(state == TIDState.CREATED || state ==
TIDState.ROLLED_BACK)
                return true;

            return false;
            // "true" means that JCo will now execute the
            // transaction, "false" means that we have already
            // executed this transaction previously, so JCo will
            // skip the handleRequest() step and will immediately
            // return an OK code to SAP.
        }

        public void commit(JCoServerContext serverCtx, String tid)
        {
            System.out.println("TID Handler: commit for " + tid);

            // react on commit e.g. commit on the database
            // if necessary throw a RuntimeException, if the commit
            // was not possible.

            availableTIDs.put(tid, TIDState.COMMITTED);
        }

        public void rollback(JCoServerContext serverCtx, String
tid)
        {
            System.out.println("TID Handler: rollback for " + tid);
            availableTIDs.put(tid, TIDState.ROLLED_BACK);

            // react on rollback e.g. rollback on the database.
        }

        public void confirmTID(JCoServerContext serverCtx, String
tid)
        {
            System.out.println("TID Handler: confirmTID for " +
tid);

            try
            {
            // clean up the resources.
            }
            // catch(Throwable t) {} //partner wont react on an
            // exception at this point.
            finally
            {
                availableTIDs.remove(tid);
            }
        }

        public void execute(JCoServerContext serverCtx)
        {
            String tid = serverCtx.getTID();
            if(tid != null)
            {
                System.out.println("TID Handler: execute for " +
tid);
                availableTIDs.put(tid, TIDState.EXECUTED);
            }
        }

        private enum TIDState
        {
            CREATED, EXECUTED, COMMITTED, ROLLED_BACK, CONFIRMED;
        }
    }

    static void step3SimpleTRfcServer()
    {
        JCoServer server;
        try
        {
            server = JCoServerFactory.getServer(SERVER_NAME1);
        }
        catch(JCoException ex)
        {
            throw new RuntimeException("Unable to create the server
" + SERVER_NAME1 + ", because of " + ex.getMessage(), ex);
        }

        JCoServerFunctionHandler stfcConnectionHandler = new
StfcConnectionHandler();
        DefaultServerHandlerFactory.FunctionHandlerFactory factory
```

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

        // in addition to step 1
        myTIDHandler = new MyTIDHandler();
        server.setTIDHandler(myTIDHandler);

        server.start();
        System.out.println("The program can be stopped using
<ctrl>+<c>");
    }

    public static void main(String[] a)
    {
        // step1SimpleServer();
        step2SimpleServer();
        // step3SimpleTRfcServer();
    }
}
```

bgRFC Calls

For bgRFC, the implementation could look like this:

For bgRFC communication, you have to specify *basXML* as transfer protocol in the corresponding RFC Destination (transaction SM59).

Processing a bgRFC Call

```
    static class MyUnitIDHandler implements JCoServerUnitIDHandler
    {
        Map<String, UnitIDState> availableUnitIDs = new
Hashtable<String, UnitIDState>();

        public boolean checkUnitID(JCoServerContext serverCtx,
JCoUnitIdentifier unitIdentifier)
        {
            // This example uses a Hashtable to store status
information. Normally, however,
            // you would use a database. If the DB is down throw a
```

```
RuntimeException at
            // this point. JCo will then abort the tRFC and the R/3
backend will try
            // again later.

            System.out.println("Unit ID Handler: checkUnitID for "
+ unitIdentifier.toString());
            UnitIDState state =
availableUnitIDs.get(unitIdentifier.getID());
            if(state == null)
            {
            availableUnitIDs.put(unitIdentifier.getID(),
UnitIDState.CREATED);
                return true;
            }

            if(state == UnitIDState.CREATED || state ==
UnitIDState.ROLLED_BACK)
                return true;

            return false;
            // "true" means that JCo will now execute the
transaction, "false" means
            // that we have already executed this transaction
previously, so JCo will
            // skip the handleRequest() step and will immediately
return an OK code to R/3.
        }

             public void confirmUnitID(JCoServerContext serverCtx,
JCoUnitIdentifier unitIdentifier)
        {
            System.out.println("Unit ID Handler: confirmTID for " +
unitIdentifier.toString());

            try
            {
                // clean up the resources
            }
            // catch(Throwable t) {} //partner won't react on an
exception at
            // this point
            finally
            {
                availableUnitIDs.remove(unitIdentifier.getID());
            }
        }

         public void commit(JCoServerContext serverCtx, 
JCoUnitIdentifier unitIdentifier)
        {
            System.out.println("Unit ID Handler: commit for " + 
unitIdentifier.toString());

            // react on commit, e.g. commit on the database;
            // if necessary throw a RuntimeException, if the commit 
was not possible
            availableUnitIDs.put(unitIdentifier.getID(), 
UnitIDState.COMMITTED);
        }

         public void rollback(JCoServerContext serverCtx, 
JCoUnitIdentifier unitIdentifier)
        {
            System.out.println("Unit ID Handler: rollback for " + 
unitIdentifier.toString());
            availableUnitIDs.put(unitIdentifier.getID(), 
UnitIDState.ROLLED_BACK);

            // react on rollback, e.g. rollback on the database
        }

         public JCoFunctionUnitState 
getFunctionUnitState(JCoServerContext serverCtx, JCoUnitIdentifier 
unitIdentifier)
        {
                    UnitIDState currentState = 
availableUnitIDs.get(unitIdentifier.getID());
                    if(currentState != null)
                           return JCoFunctionUnitState.NOT_FOUND;

                    switch(currentState)
                    {
                           case CREATED:
                           case EXECUTED:
                                  return 
JCoFunctionUnitState.IN_PROCESS;
                           case COMMITTED:
                                  return
JCoFunctionUnitState.COMMITTED;
                           case ROLLED_BACK:
                                  return
JCoFunctionUnitState.ROLLED_BACK;
                           case CONFIRMED:
                                  return
JCoFunctionUnitState.CONFIRMED;
                    }
               return JCoFunctionUnitState.NOT_FOUND;
        }

        public void execute(JCoServerContext serverCtx)
        {
            JCoUnitIdentifier unitIdentifier =
serverCtx.getUnitIdentifier();
            if(unitIdentifier != null)
            {
                System.out.println("Unit ID Handler: execute for "
+ unitIdentifier.toString());
                availableUnitIDs.put(unitIdentifier.getID(),
UnitIDState.EXECUTED);
            }
        }

        private enum UnitIDState
        {
            CREATED, EXECUTED, COMMITTED, ROLLED_BACK, CONFIRMED;
        }
    }
```

Example Program JCo Server

The example below contains a complete JCo server program. It is made up of the code examples from the activities described in the previous sections:

- Java Programm for Establishing a Server Connection
- Implementing an Exception Listener
- Monitoring Server Connections
- Processing an ABAP Call

JCo Server Program

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.Hashtable;
import java.util.Map;
import java.util.Properties;

import com.sap.conn.jco.JCoException;
import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.ext.DestinationDataProvider;
import com.sap.conn.jco.ext.ServerDataProvider;
import com.sap.conn.jco.server.DefaultServerHandlerFactory;
import com.sap.conn.jco.server.JCoServer;
import com.sap.conn.jco.server.JCoServerContext;
import com.sap.conn.jco.server.JCoServerErrorListener;
import com.sap.conn.jco.server.JCoServerExceptionListener;
import com.sap.conn.jco.server.JCoServerFactory;
import com.sap.conn.jco.server.JCoServerFunctionHandler;
import com.sap.conn.jco.server.JCoServerState;
import com.sap.conn.jco.server.JCoServerStateChangedListener;
import com.sap.conn.jco.server.JCoServerTIDHandler;


public class StepByStepServer
{
    static String SERVER_NAME1 = "SERVER";
    static String DESTINATION_NAME1 = "ABAP_AS_WITHOUT_POOL";
    static String DESTINATION_NAME2 = "ABAP_AS_WITH_POOL";
    static
    {
        Properties connectProperties = new Properties();

connectProperties.setProperty(DestinationDataProvider.JCO_ASHOST,
"ls4065");

connectProperties.setProperty(DestinationDataProvider.JCO_SYSNR,
"85");

connectProperties.setProperty(DestinationDataProvider.JCO_CLIENT,
"800");
```

```
connectProperties.setProperty(DestinationDataProvider.JCO_USER,
"homo faber");

connectProperties.setProperty(DestinationDataProvider.JCO_PASSWD,
"alaska");

connectProperties.setProperty(DestinationDataProvider.JCO_LANG,
"en");
        createDataFile(DESTINATION_NAME1, "jcoDestination",
connectProperties);



connectProperties.setProperty(DestinationDataProvider.JCO_POOL_CAPA
CITY, "3");

connectProperties.setProperty(DestinationDataProvider.JCO_PEAK_LIMI
T,    "10");
        createDataFile(DESTINATION_NAME2, "jcoDestination",
connectProperties);


        Properties servertProperties = new Properties();

servertProperties.setProperty(ServerDataProvider.JCO_GWHOST,
"binmain");

servertProperties.setProperty(ServerDataProvider.JCO_GWSERV,
"sapgw53");

servertProperties.setProperty(ServerDataProvider.JCO_PROGID,
"JCO_SERVER");

servertProperties.setProperty(ServerDataProvider.JCO_REP_DEST,
"ABAP_AS_WITH_POOL");

servertProperties.setProperty(ServerDataProvider.JCO_CONNECTION_COU
NT, "2");
        createDataFile(SERVER_NAME1, "jcoServer",
servertProperties);
    }


    static void createDataFile(String name, String suffix,
Properties properties)
    {
        File cfg = new File(name+"."+suffix);
        if(!cfg.exists())
        {
            try
            {
                FileOutputStream fos = new FileOutputStream(cfg,
false);
                properties.store(fos, "for tests only !");
                fos.close();
            }
            catch (Exception e)
            {
                throw new RuntimeException("Unable to create the
destination file " + cfg.getName(), e);
            }
        }
    }

    static class StfcConnectionHandler implements
JCoServerFunctionHandler
    {
        public void handleRequest(JCoServerContext serverCtx,
JCoFunction function)
        {
            System.out.println("----------------------------------
-----------------------------");
            System.out.println("call              : " +
function.getName());
            System.out.println("ConnectionId      : " +
serverCtx.getConnectionID());
            System.out.println("SessionId         : " +
serverCtx.getSessionID());
            System.out.println("TID               : " +
serverCtx.getTID());
            System.out.println("repository name   : " +
serverCtx.getRepository().getName());
            System.out.println("is in transaction : " +
serverCtx.isInTransaction());
            System.out.println("is stateful       : " +
serverCtx.isStatefulSession());
            System.out.println("----------------------------------
-----------------------------");
            System.out.println("gwhost: " +
serverCtx.getServer().getGatewayHost());
            System.out.println("gwserv: " +
serverCtx.getServer().getGatewayService());
            System.out.println("progid: " +
serverCtx.getServer().getProgramID());
            System.out.println("----------------------------------
-----------------------------");
```

```
            System.out.println("attributes  : ");

System.out.println(serverCtx.getConnectionAttributes().toString());
            System.out.println("----------------------------------
------------------------------");
            System.out.println("req text: " +
function.getImportParameterList().getString("REQUTEXT"));
            function.getExportParameterList().setValue("ECHOTEXT",
function.getImportParameterList().getString("REQUTEXT"));
            function.getExportParameterList().setValue("RESPTEXT",
"Hello World");
        }
    }

    static void step1SimpleServer()
    {
        JCoServer server;
        try
        {
            server = JCoServerFactory.getServer(SERVER_NAME1);
        }
        catch(JCoException ex)
        {
            throw new RuntimeException("Unable to create the server
" + SERVER_NAME1 + ", because of " + ex.getMessage(), ex);
        }

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

        server.start();
        System.out.println("The program can be stopped using
<ctrl>+<c>");
    }

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

    static class MyStateChangedListener implements
JCoServerStateChangedListener
    {
        public void serverStateChangeOccurred(JCoServer server,
JCoServerState oldState, JCoServerState newState)
        {

            // Defined states are: STARTED, DEAD, ALIVE, STOPPED;
            // see JCoServerState class for details.
            // 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());
        }
    }

    static void step2SimpleServer()
    {
        JCoServer server;
        try
        {
            server = JCoServerFactory.getServer(SERVER_NAME1);
        }
        catch(JCoException ex)
        {
            throw new RuntimeException("Unable to create the server
" + SERVER_NAME1 + ", because of " + ex.getMessage(), ex);
        }

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

        //additionally to step 1
        MyThrowableListener eListener = new MyThrowableListener();
        server.addServerErrorListener(eListener);
        server.addServerExceptionListener(eListener);

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

        server.start();
        System.out.println("The program can be stopped using
<ctrl>+<c>");
    }

    static class MyTIDHandler implements JCoServerTIDHandler
    {

        Map<String, TIDState> availableTIDs = new Hashtable<String,
TIDState>();

        public boolean checkTID(JCoServerContext serverCtx, String
tid)
        {
            System.out.println("TID Handler: checkTID for " + tid);
            if(availableTIDs.containsKey(tid))
                return false;

            //notify other instances about the starting of a
tRFC/bgRFC call, if necessary
            availableTIDs.put(tid, TIDState.STARTED);
            return true;
        }

        public void commit(JCoServerContext serverCtx, String tid)
        {
            System.out.println("TID Handler: commit for " + tid);

            //react on commit e.g. commit on the database
            //if necessary throw a RuntimeException if the commit
was not possible
            availableTIDs.put(tid, TIDState.FINISHED);
        }

        public void rollback(JCoServerContext serverCtx, String
tid)
        {
            System.out.println("TID Handler: rollback for " + tid);

            //react on rollback e.g. rollback on the database
            availableTIDs.put(tid, TIDState.FINISHED);
        }

        public void confirmTID(JCoServerContext serverCtx, String
tid)
        {
            System.out.println("TID Handler: confirmTID for " +
tid);

            try
            {
                //clean up the resources
            }
            //catch(Throwable t) {} //partner wont react on an
exception at this point
            finally
            {
                availableTIDs.remove(tid);
            }
        }

        private enum TIDState
        {
            STARTED, FINISHED;
        }
    }



    static void step3SimpleTRfcServer()
    {
        JCoServer server;
        try
        {
            server = JCoServerFactory.getServer(SERVER_NAME1);
        }
        catch(JCoException ex)
        {
            throw new RuntimeException("Unable to create the server
" + SERVER_NAME1 + ", because of " + ex.getMessage(), ex);
        }

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

        //additionally to step 1
        MyTIDHandler myTIDHandler = new MyTIDHandler();
        server.setTIDHandler(myTIDHandler);

        server.start();
        System.out.println("The program can be stopped using
<ctrl>+<c>");
    }



    public static void main(String[] a)
    {
        step1SimpleServer();
        step2SimpleServer();
        step3SimpleTRfcServer();
    }
```

```
}
```

Stateful Calls

The following example shows stateful RFC function module processing in a JCo server scenario (ABAP calls Java).

Using the function modules `INCREMENT_COUNTER` and `GET_COUNTER` simple repository queries will be executed to make clear the difference between stateless and stateful calls.

In the stateful scenario the counter value will be incremented by 1 with each query, in the stateless scenario the value is always 1, since every query opens a new context.

Before you can execute this example a wrapper is needed fort the function modules `Z_INCREMENT_COUNTER` and `Z_GET_COUNTER` im AS ABAP because the existing function modules `INCREMENT_COUNTER` and `GET_COUNTER` are not remote enabled:

Wrapper for `INCREMENT_COUNTER` and `GET_COUNTER`

```
* remote enabled function Z_INCREMENT_COUNTER wrapping the
INCREMENT_COUNTER

           FUNCTION Z_INCREMENT_COUNTER.
           CALL FUNCTION 'INCREMENT_COUNTER'.
           ENDFUNCTION.

   * remote enabled function Z_GET_COUNTER wrapping the GET_COUNTER

           FUNCTION Z_GET_COUNTER.
           CALL FUNCTION 'GET_COUNTER'
               IMPORTING
                  GET_VALUE = GET_VALUE
           .
           ENDFUNCTION.

    * with GET_VALUE TYPE  I as export parameter and report
ZJCO_STATEFUL_COUNTER

                  REPORT  ZJCO_STATEFUL_COUNTER.
           PARAMETER dest TYPE RFCDEST.

           DATA value TYPE i.
           DATA loops TYPE i VALUE 5.
           DO loops TIMES.
               CALL FUNCTION 'Z_INCREMENT_COUNTER' DESTINATION dest.
           ENDDO.

           CALL FUNCTION 'Z_GET_COUNTER' DESTINATION dest
               IMPORTING
                  GET_VALUE       = value
           .
            IF value <> loops.
             write: / 'Error expecting ', loops, ', but get ', value,
' as counter value'.
           ELSE.
             write: / 'success'.
           ENDIF.
```

In a next step the calls will be processed:

Stateful Calls (Server Scenario)

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.Hashtable;
import java.util.Map;
import java.util.Properties;

import com.sap.conn.jco.JCoException;
import com.sap.conn.jco.JCoFunction;
import com.sap.conn.jco.ext.DestinationDataProvider;
import com.sap.conn.jco.ext.ServerDataProvider;
import com.sap.conn.jco.server.JCoServer;
import com.sap.conn.jco.server.JCoServerContext;
import com.sap.conn.jco.server.JCoServerFactory;
import com.sap.conn.jco.server.JCoServerFunctionHandler;
import com.sap.conn.jco.server.JCoServerFunctionHandlerFactory;

public class StatefulServerExample
{
    static String SERVER = "SERVER";
    static String ABAP_AS_WITHOUT_POOL = "ABAP_AS_WITHOUT_POOL";
    static
    {
        Properties connectProperties = new Properties();

connectProperties.setProperty(DestinationDataProvider.JCO_ASHOST,
"binmain");

connectProperties.setProperty(DestinationDataProvider.JCO_SYSNR,
"53");

connectProperties.setProperty(DestinationDataProvider.JCO_CLIENT,
"000");

connectProperties.setProperty(DestinationDataProvider.JCO_USER,
"JCOTEST");

connectProperties.setProperty(DestinationDataProvider.JCO_PASSWD,
"JCOTEST");

connectProperties.setProperty(DestinationDataProvider.JCO_LANG,
"en");
        createDataFile(ABAP_AS_WITHOUT_POOL, "jcoDestination",
connectProperties);


        Properties servertProperties = new Properties();
        servertProperties.setProperty(ServerDataProvider.JCO_GWHOST,
"binmain");
        servertProperties.setProperty(ServerDataProvider.JCO_GWSERV,
"sapgw53");
        servertProperties.setProperty(ServerDataProvider.JCO_PROGID,
"JCO_SERVER");
        servertProperties.setProperty(ServerDataProvider.JCO_REP_DEST,
ABAP_AS_WITHOUT_POOL);

servertProperties.setProperty(ServerDataProvider.JCO_CONNECTION_COUNT,
"2");
        createDataFile(SERVER, "jcoServer", servertProperties);
    }


    static void createDataFile(String name, String suffix, Properties
properties)
    {
        File cfg = new File(name+"."+suffix);
        if(!cfg.exists())
        {
            try
            {
                FileOutputStream fos = new FileOutputStream(cfg,
false);
```

```
                properties.store(fos, "for tests only !");
                fos.close();
            }
            catch (Exception e)
            {
                throw new RuntimeException("Unable to create the
destination file " + cfg.getName(), e);
            }
        }
    }



    static class MyFunctionHandlerFactory implements
JCoServerFunctionHandlerFactory
    {
        class SessionContext
        {
            Hashtable<String, Object> cachedSessionData = new
Hashtable<String, Object>();
        }

        private Map<String, SessionContext> statefulSessions =
            new Hashtable<String, SessionContext>();

        private ZGetCounterFunctionHandler zGetCounterFunctionHandler
=
            new ZGetCounterFunctionHandler();

        private ZIncrementCounterFunctionHandler
zIncrementCounterFunctionHandler =
            new ZIncrementCounterFunctionHandler();

        public JCoServerFunctionHandler
getCallHandler(JCoServerContext serverCtx, String functionName)
        {
            JCoServerFunctionHandler handler = null;

            if(functionName.equals("Z_INCREMENT_COUNTER"))
                handler = zIncrementCounterFunctionHandler;
            else if(functionName.equals("Z_GET_COUNTER"))
                handler = zGetCounterFunctionHandler;
            if(handler instanceof StatefulFunctionModule)
            {
                SessionContext cachedSession;
                if(!serverCtx.isStatefulSession())
                {
                    serverCtx.setStateful(true);
                    cachedSession = new SessionContext();
                    statefulSessions.put(serverCtx.getSessionID(),
cachedSession);
                }
                else
                {
                    cachedSession =
statefulSessions.get(serverCtx.getSessionID());
                    if(cachedSession == null)
                        throw new RuntimeException("Unable to find the
session context for session id " + serverCtx.getSessionID());
                }

((StatefulFunctionModule)handler).setSessionData(cachedSession.cachedS
essionData);
                return handler;
            }

            //null leads to a system failure on the ABAP side
            return null;
        }

        public void sessionClosed(JCoServerContext serverCtx, String
message, boolean error)
        {
            System.out.println("Session " + serverCtx.getSessionID() +
" was closed " + (error?message:"by SAP system"));
            statefulSessions.remove(serverCtx.getSessionID());
        }
    }

    static abstract class StatefulFunctionModule implements
JCoServerFunctionHandler
    {
        Hashtable<String, Object> sessionData;
        public void setSessionData(Hashtable<String, Object>
sessionData)
```

```
        {
            this.sessionData = sessionData;
        }
    }



    static class ZGetCounterFunctionHandler extends
StatefulFunctionModule
    {
        public void handleRequest(JCoServerContext serverCtx,
JCoFunction function)
        {
            System.out.println("ZGetCounterFunctionHandler: return
counter");
            Integer counter = (Integer)sessionData.get("COUNTER");
            if(counter == null)

function.getExportParameterList().setValue("GET_VALUE", 0);
            else

function.getExportParameterList().setValue("GET_VALUE",
counter.intValue());
        }


    }

    static class ZIncrementCounterFunctionHandler extends
StatefulFunctionModule
    {
        public void handleRequest(JCoServerContext serverCtx,
JCoFunction function)
        {
            System.out.println("ZIncrementCounterFunctionHandler:
increase counter");
            Integer counter = (Integer)sessionData.get("COUNTER");
            if(counter == null)
                sessionData.put("COUNTER", new Integer(1));
            else
                sessionData.put("COUNTER", new
Integer(counter.intValue()+1));
        }
    }
    public static void main(String[] args)
    {
        JCoServer server;
        try
        {
            server = JCoServerFactory.getServer(SERVER);
        }
        catch(JCoException ex)
        {
            throw new RuntimeException("Unable to create the server "
+ SERVER + ", because of " + ex.getMessage(), ex);
        }

        server.setCallHandlerFactory(new MyFunctionHandlerFactory());

        server.start();
        System.out.println("The program can be stopped using
<ctrl>+<c>");
    }
}
```

IDoc Support for external Java Applications

SAP JCo 3.0 can be used with the IDoc class library 3.0 that supports the IDoc-based communication of external (non-SAP) Java applications with the AS ABAP.

Funktionsumfang

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 separate software component that can be downloaded in addition to SAP JCo 3.0 separately from SAP Service Marketplace (`service.sap.com/connectors`).

In contrast with the former versions the current IDoc class library 3.0 is based – like SAP JCo 3.0 – 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.

Below you will find the following example programs for the IDoc class library:

- Example program IDoc client
- Example program IDoc server

Example Program IDoc Client

The following example program shows the use of the IDoc class library for IDoc communication via SAP JCo 3.0 acting as a client.

IDoc Client

```
package com.sap.conn.idoc.examples;


import com.sap.conn.jco.*;

import com.sap.conn.idoc.jco.*;

import com.sap.conn.idoc.*;

import java.io.*;


public class IDocClientExample {


    public static void main(String[] args) {


        try

     {

            String iDocXML = null;

            FileReader fileReader;


            try
            {
                fileReader = new FileReader("MyIDocDocumentAsXML.xml");
                BufferedReader br = new BufferedReader(fileReader);
                StringBuffer sb = new StringBuffer();
                String line;
                while ((line = br.readLine()) != null)
                {
                    sb.append(line);
                }
                iDocXML = sb.toString();



                br.close();
                fileReader.close();
            }
            catch(Exception ex)
            {
                ex.printStackTrace();
            }

            // see configuration file BCE.jcoDestination provided in the installation directory.
        JCoDestination destination=JCoDestinationManager.getDestination("BCE");
        IDocRepository iDocRepository = JCoIDoc.getIDocRepository(destination);
        String tid = destination.createTID();
        IDocFactory iDocFactory = JCoIDoc.getIDocFactory();

        // a) create new idoc
        IDocDocument doc = iDocFactory.createIDocDocument(iDocRepository,
"MATMAS02");
        IDocSegment segment = doc.getRootSegment();
        segment = segment.addChild("E1MARAM");
        // and so on. See IDoc Specification .....
        JCoIDoc.send(doc, IDocFactory.IDOC_VERSION_DEFAULT, destination, tid);
```

```
            // b) use existent xml file
            IDocXMLProcessor processor=iDocFactory.getIDocXMLProcessor();
           IDocDocumentList iDocList=processor.parse(iDocRepository, iDocXML);
           JCoIDoc.send(iDocList, IDocFactory.IDOC_VERSION_DEFAULT, destination, tid);
           destination.confirmTID(tid);

        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        System.out.print("program end");
      }
}
```

Example Program IDoc Server

The following example program shows the use of the IDoc class library for IDoc communication via SAP JCo 3.0 acting as a server.

IDoc Server

```
package com.sap.conn.idoc.examples;


import com.sap.conn.idoc.*;


import java.io.*;
import com.sap.conn.jco.server.*;
import com.sap.conn.idoc.jco.*;



public class IDocServerExample
{
    public static void main(String[] a)
    {
        try
        {
             // see examples of configuration files MYSERVER.jcoServer and
BCE.jcoDestination provided in the installation directory.
             JCoIDocServer server = JCoIDoc.getServer("MYSERVER");
             server.setIDocHandlerFactory(new MyIDocHandlerFactory());
             server.setTIDHandler(new MyTidHandler());

             MyThrowableListener listener = new MyThrowableListener();
             server.addServerErrorListener(listener);
            server.addServerExceptionListener(listener);
            server.setConnectionCount(1);
            server.start();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
     }

    static class MyIDocHandler implements JCoIDocHandler
    {
        public void handleRequest(JCoServerContext serverCtx, IDocDocumentList idocList)
        {

             FileOutputStream fos=null;
            OutputStreamWriter osw=null;
             try
             {
                   IDocXMLProcessor xmlProcessor =
                         JCoIDoc.getIDocFactory().getIDocXMLProcessor();
               fos=new FileOutputStream(serverCtx.getTID()+"_idoc.xml");
               osw=new OutputStreamWriter(fos, "UTF8");
```

```
                xmlProcessor.render(idocList, osw,
                          IDocXMLProcessor.RENDER_WITH_TABS_AND_CRLF);
                osw.flush();
            }
            catch (Throwable thr)
            {
                thr.printStackTrace();
            }
           finally
           {
              try
              {
                 if (osw!=null)
                    osw.close();
                 if (fos!=null)
                    fos.close();
              }
              catch (IOException e)
              {
                 e.printStackTrace();
              }
           }
        }
    }

    static class MyIDocHandlerFactory implements JCoIDocHandlerFactory
    {
      private JCoIDocHandler handler = new MyIDocHandler();
      public JCoIDocHandler getIDocHandler(JCoIDocServerContext serverCtx)
      {
            return handler;
      }
    }
    static class MyThrowableListener implements JCoServerErrorListener,
JCoServerExceptionListener
    {

        public void serverErrorOccurred(JCoServer server, String connectionId, Error error)
        {
            System.out.println(">>> Error occured on " + server.getProgramID() + " connection "
+ connectionId);
            error.printStackTrace();
        }
        public void serverExceptionOccurred(JCoServer server, String connectionId,
Exception error)
        {
            System.out.println(">>> Error occured on " + server.getProgramID() + " connection "
+ connectionId);
            error.printStackTrace();
        }
    }

    static class MyTidHandler implements JCoServerTIDHandler
    {
      public boolean checkTID(JCoServerContext serverCtx, String tid)
      {
           System.out.println("checkTID called for TID="+tid);
           return true;
      }


      public void confirmTID(JCoServerContext serverCtx, String tid)
      {
           System.out.println("confirmTID called for TID="+tid);
      }


      public void commit(JCoServerContext serverCtx, String tid)
```

```
        {
            System.out.println("commit called for TID="+tid);
        }



        public void rollback(JCoServerContext serverCtx, String tid)
        {
            System.out.print("rollback called for TID="+tid);
        }
    }
}
```