RFC Java Class Library (BC-FES-AIT)

Release 4.6C
Copyright

© Copyright 2001 SAP AG. All rights reserved.

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

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

Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.

IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.

ORACLE® is a registered trademark of ORACLE Corporation.

INFORMIX®-OnLine for SAP and Informix® Dynamic Server™ are registered trademarks of Informix Software Incorporated.

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

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

JAVA® is a registered trademark of Sun Microsystems, Inc.

JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

SAP, SAP Logo, R/2, R/3, ABAP, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.
## Icons

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚠️</td>
<td>Caution</td>
</tr>
<tr>
<td>✍️</td>
<td>Example</td>
</tr>
<tr>
<td>💡</td>
<td>Note</td>
</tr>
<tr>
<td>🎯</td>
<td>Recommendation</td>
</tr>
<tr>
<td>🔧</td>
<td>Syntax</td>
</tr>
</tbody>
</table>
Contents

RFC Java Class Library (BC-FES-AIT) ..............................................................6
Java RFC Architecture .............................................................................................. 7
Middleware Used by Java RFC ........................................................................................ 9
  CORBA-type Middleware: Orbix .............................................................................. 10
  SAP JNI Middleware ............................................................................................... 12
Java RFC Features ........................................................................................................ 14
Java RFC Documentation ............................................................................................. 18
Release Information ....................................................................................................... 19
What’s New in Release 4.6A? ...................................................................................... 20
What’s New in Release 4.6B? ...................................................................................... 21
What’s New in Release 4.6C? ...................................................................................... 22
Installation and Setup ................................................................................................. 23
Installing Java RFC for JNI Middleware ........................................................................ 24
  Orbix Installation and Setup ...................................................................................... 25
    Installing the Orbix Client ...................................................................................... 27
    Installing the Orbix Server ..................................................................................... 28
    Starting the Orbix Daemon ..................................................................................... 30
    Changing the Orbix Daemon Setup After Installation ............................................. 31
    Configuring the Java RFC Orbix Server ................................................................. 32
    Java RFC Orbix Server Installation Processing ..................................................... 34
    Starting the Java RFC Orbix Server ....................................................................... 36
The SAP Java RFC Program Group ........................................................................... 38
RFC Java Library Programming Guide ........................................................................ 39
Running Programs that Use JDK 1.2 and Java RFC ................................................... 40
Interface for Building Client Applications .................................................................. 41
  Java RFC Client Files and Objects .......................................................................... 42
    The SessionManager and SessionInfo Objects .................................................. 43
    The Java RFC Properties Files ............................................................................. 45
    The Function Module Object (IRfcModule) .......................................................... 51
  Parameter and Field Interfaces ................................................................................. 53
    Parameter and Field Metadata Classes ............................................................... 58
    Table, Row, and Cursor Interfaces ....................................................................... 61
  Factory Objects ......................................................................................................... 62
Specifying Middleware Type ........................................................................................ 64
Using the Client Interface to Make an RFC Call .......................................................... 66
Examples ..................................................................................................................... 70
  Example: Calling RFC Functions with No Parameters .......................................... 71
  Example: RFC Function with an Export Parameter .............................................. 72
  Example: Calling an RFC Function with Parameters .......................................... 73
  Additional Java RFC Client Samples .................................................................... 77
Setting Up and Starting a Session ................................................................................ 80
  Using the Properties Files to Set Up SessionInfo ............................................... 82
  Adding or Changing Connection Properties Used .............................................. 83

April 2001
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually Setting the Properties of SessionInfo</td>
<td>85</td>
</tr>
<tr>
<td>Setting Up the Function Module Object</td>
<td>86</td>
</tr>
<tr>
<td>Auto-Creating the IRfcModule Object</td>
<td>89</td>
</tr>
<tr>
<td>Manually Creating the IRfcModule Object</td>
<td>91</td>
</tr>
<tr>
<td>Calling an RFC Function Module</td>
<td>93</td>
</tr>
<tr>
<td>Reading the Data from Export Parameters</td>
<td>94</td>
</tr>
<tr>
<td>Working with Table Data</td>
<td>96</td>
</tr>
<tr>
<td>Interface for Building Server Applications</td>
<td>100</td>
</tr>
<tr>
<td>Java RFC Server Classes and Interfaces</td>
<td>101</td>
</tr>
<tr>
<td>Building Java RFC Server Applications</td>
<td>102</td>
</tr>
<tr>
<td>Server Programming with Manual Creation of Function Objects</td>
<td>104</td>
</tr>
<tr>
<td>Server Programming with Automatic Creation of Function Objects</td>
<td>107</td>
</tr>
<tr>
<td>Obtaining SessionManager</td>
<td>110</td>
</tr>
<tr>
<td>Obtaining the ServerApp Object</td>
<td>111</td>
</tr>
<tr>
<td>Using the IServerFunctionFactory Object</td>
<td>112</td>
</tr>
<tr>
<td>Using the Parameter Factory Objects to Add and Create Parameters</td>
<td>113</td>
</tr>
<tr>
<td>Adding Parameter Objects to IServerFunction</td>
<td>115</td>
</tr>
<tr>
<td>Creating a Server Process for Incoming Calls</td>
<td>116</td>
</tr>
<tr>
<td>Associating IServerFunction and IServerProcess</td>
<td>117</td>
</tr>
<tr>
<td>Adding IServerFunction to ServerApp</td>
<td>118</td>
</tr>
<tr>
<td>Registering a Server Application at the SAP Gateway</td>
<td>119</td>
</tr>
<tr>
<td>Adjusting the Number of Server Threads</td>
<td>121</td>
</tr>
<tr>
<td>Running a Server Application</td>
<td>122</td>
</tr>
<tr>
<td>Server Sample Programs: Srfctest and SrfctestAuto</td>
<td>123</td>
</tr>
<tr>
<td>Advanced Topics</td>
<td>125</td>
</tr>
<tr>
<td>Programming Multiple Client Connections</td>
<td>126</td>
</tr>
<tr>
<td>Classes and Interfaces for Multiple Connections</td>
<td>127</td>
</tr>
<tr>
<td>FactoryManager</td>
<td>128</td>
</tr>
<tr>
<td>IRfcConnection and Related Objects</td>
<td>129</td>
</tr>
<tr>
<td>IRfcModule in the Context of a Connection</td>
<td>131</td>
</tr>
<tr>
<td>Handling Multiple Connections</td>
<td>133</td>
</tr>
<tr>
<td>Setting Up the Factory Manager</td>
<td>137</td>
</tr>
<tr>
<td>Multi-Threading in RFC Client Applications</td>
<td>138</td>
</tr>
<tr>
<td>Handling Exceptions and Errors</td>
<td>139</td>
</tr>
</tbody>
</table>
RFC Java Class Library (BC-FES-AIT)

Purpose

The Java RFC Class Library provides a simple, object-oriented Java view of the SAP RFC (Remote Function Call) API.

Based on the RFC Library (which provides the RFC API), the Java RFC Class Library allows you to easily write Java programs that use the RFC protocol to communicate with R/3 applications. The Java RFC Class Library hides all the low-level details of the native C or C++ calls, thereby reducing much of the complexity for programming RFC in Java.

The Java RFC Class Library allows you to develop both RFC client and RFC server applications. This means that applications you develop with the Java RFC Class Library can either send client requests to R/3 using RFC calls, or they can act as a RFC servers to R/3 function calls.

Note that we sometimes refer to the Java RFC Class Library in short as the Java RFC.

Implementation Considerations

You can use the Java RFC Class Library with an R/3 system 2.x and higher if the R/3 system had been set up to send or receive remotely callable functions.
Java RFC Architecture

Underlying Software

The Java RFC is based on the RFC Library (also called the RFC API). The Java RFC packages in some cases also use the framework provided by the RFC C++ Class Library [Ext.] (which is a part of the SAP Automation suite of products [Ext.]), to take advantage of the object-oriented paradigm of the RFC C++ Class Library.

Java RFC Class Library

The Java RFC Class Library provides packages that include interfaces and classes for creating RFC client or RFC server applications.

The Java RFC Class Library is structured as two layers:

- The Java RFC Interface layer (fixed)
  
  This layer contains the definition of the interfaces and classes that you use when programming with the Java RFC Class Library. This layer stays consistent, in that the definition of existing classes and interfaces does not change. SAP may add interfaces or classes over time.

  This layer also contains the implementation of the classes that are a part of the Java RFC interface definition.

- Middleware implementation layer (pluggable)

  This layer contains the implementation of the interfaces that are a part of the Java RFC interface definition layer.

  The middleware implementation layer is pluggable, meaning that the Java RFC Class Library allows you to interchange middleware implementations. SAP provides two different types of middleware implementations for you to use with the Java RFC Class Library. Other companies provide the middleware implementation for the Java RFC Class Library as well.

The following diagram shows the components used by the Java RFC.
The *Java RFC Class Library* allows you to choose between two types of provided middleware:

- [CORBA (Orbix)](Page 10)
- [Java Native Interface (JNI)](Page 12)

**See Also**

Middleware Used by Java RFC [Page 9]
Middleware Used by Java RFC

Middleware Types

The Java RFC Class Library allows you to choose between two types of middleware:

- CORBA
- Java Native Interface (JNI)

SAP provides two middleware implementations with the Java RFC Class Library (one of each of those two type):

<table>
<thead>
<tr>
<th>Middleware Type</th>
<th>Bundled Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>IONA Technologies’ Orbix</td>
</tr>
<tr>
<td>JNI</td>
<td>SAP’s JNI</td>
</tr>
</tbody>
</table>

You choose which of the middleware implementation you wish to use with the Java RFC Class Library.

Comparing Middleware Types

Using the CORBA middleware implementation that uses IONA Technologies’ Orbix allows you to distribute the RFC application across the network. This means that you can use it to program Java applets that make RFC calls to R/3.

The JNI middleware implementation does not allow you to distribute your RFC application to remote client computers. This means that you cannot use it to implement Java applets that make RFC calls.

On the other hand, the JNI middleware is easier to use, and it is more efficient to use if all the RFC components of your RFC application reside on the same computer.

This means that the JNI implementation is appropriate for creating Web server-side applications that make RFC calls to R/3.

See Also

For a more detailed explanation of the differences between the architecture of the Java RFC components when using each of these middleware implementations, see the following topics:

- CORBA-type Middleware: Orbix [Page 10]
- SAP JNI Middleware [Page 12]
CORBA-type Middleware: Orbix

The Java RFC Class Library offers a CORBA implementation of the middleware using IONA Technologies’ Orbix 2.3c and OrbixWeb 3.0.

The following diagram shows the different components used by the Java RFC Class Library when using the Orbix middleware. The arrows show the flow of communication from your application all the way to the R/3 Application server and back.

Notes on the above illustration:
- Among the components shown, the Java RFC Packages, Java RFC Proxy Objects, and the C++ Orbix Server Objects are part of the Java RFC product. The Orbix components and the C++ Class Library are also bundled with the Java RFC.
- Your Java Client or Server Application is the program that uses Java RFC to communicate with R/3.
  - In the case of a Java application it can be an RFC client application (meaning that it issues RFC function calls to an R/3 application) or it can be an RFC server application (meaning that it can accept and process RFC function calls from an R/3 application).
  - In the case of Java applets, it can only be an RFC client application.
In the case of Java applets, the Orbix server must run on the same machine as the Web server, due to restrictions on applet operations enforced by Web browsers.

- The dashed line in the diagram is to indicate that the Orbix client packages create proxy objects for the “real” objects on the Orbix server. The proxy objects operate on behalf of the real objects transparently to the client.
- The communication between the Orbix client and Orbix server is through Common Object Request Broker Architecture (CORBA) interfaces. The ORB (Object Request Broker) used are Orbix and OrbixWeb from IONA Technologies.

Java RFC uses Orbix and OrbixWeb to facilitate cross network communications between Orbix client and Orbix server machines, and for hiding marshaling and networking details from your application (marshaling and unmarshaling are the processes of packaging and interpreting network messages).

The necessary Orbix and OrbixWeb runtime files are bundled with Java RFC.

As you can see from the above illustration, both the Orbix and the Java RFC products are composed of a client and a server portions, which can be installed on separate computers.

This division between the client and server portions of the Orbix middleware implementation allows you to distribute the RFC application across the network.

This division allows multiple applications to share a single RFC server. In this scenario, the Orbix server portion is installed on one computer, and the client portion can be installed on multiple clients.

This division also allows the use of Java applets that make RFC calls to R/3. In this scenario, the Java applet resides on the Web server computer. When a Web Browser (on the client computer) calls an applet, the Web server ships the applet to the client computer. The applet runs on the Web client computer. The Orbix software takes care of the communication over the network between the Web client and the Web Server when the applet makes RFC calls.

Note on the Use of the Client and Server Terms

Note that there may be a confusion with the use of the client and server terminology when using the Orbix middleware with the Java RFC Class Library. The terms client and server are used in two contexts:

- When using the Orbix middleware, both the Orbix software and the Java RFC Class Library consist of a client and a server portion. This is what allows you to distribute your application.
  
  In this context we refer to these terms as the “Orbix client” or "Orbix server".
- The whole purpose of using the Java RFC Class Library is to create either an RFC client or an RFC server application, relative to RFC calls to R/3 applications.
  
  In this context we refer to these terms as the “RFC server” or "RFC client".

Note that we use the full terms ("RFC client", for example, instead of "client") only when confusion may arise and only when it does not make the discussion cumbersome.

See Also

SAP JNI Middleware [Page 12], Middleware used by Java RFC [Page 9]
SAP JNI Middleware

The following diagram shows the components used by the Java RFC Class Library when using the JNI middleware.

The arrows show the flow of communication from your application all the way to the R/3 Application server and back.

### Notes on the above illustration:
- **Your Java Client or Server Application** represents the program that uses the Java RFC to communicate with R/3. It can be an RFC client application (meaning that it issues RFC function calls to an R/3 application) or it can be an RFC server application (meaning that it can accept RFC function calls from an R/3 application).
  - It cannot be a Java applets
- The **Java RFC Class Library** layer represents the exposed interfaces and their implementation classes of the Java RFC packages.
- The **Java RFC packages** encapsulate the RFC API and the RFC C++ Class Library. The RFC C++ Class Library is bundled with Java RFC. The RFC API is part of the SAP Presentation CD.
- The **Java RFC Native DLL** component is the SAP middleware implementation of JNI. It is also bundled with the Java RFC.

As you can see from the above illustration, all the components that participate in providing the Java RFC interfaces when using the JNI middleware implementation reside on the same computer.
Therefore, the JNI middleware implementation does not allow for distributed computing. You cannot distribute your RFC application to different computers, and you cannot use it to implement Java applets that make RFC calls.

On the other hand, the JNI middleware is easier to use. Because less communications needs to take place between various internal components (as with the arrangement of the Orbix client and server components), the JNI middleware implementation is more efficient to use if you do not need to distribute your application.

The JNI implementation is appropriate for creating Web server-side applications that make RFC calls to R/3. In this scenario you install all the Java RFC components on the Web server computer.

**See Also**

CORBA-type Middleware: Orbix [Page 10], Middleware used by Java RFC [Page 9]
Java RFC Features

Client Interface Features

The Java RFC Class Library offers classes and objects that allow you to write RFC client application programs that send RFC requests to R/3.

The features the Java RFC Class Library offers specifically for building RFC client applications are:

- Transparent RFC calls from Java: using this package you can easily build applications/applets that communicate with R/3 through RFC completely in Java.
- Establishing and closing down connections to R/3: All modes of RFC connection are supported.
- Handling a single connection is easy with the SessionManager [Page 43] and related objects. You can also handle multiple connections [Page 126], by using the FactoryManager and related objects.
- Handling parameters in calls to RFC:
  - Packaging all import and table parameter values into the right RFC-required format
  - Marshalling them over the network before each call
  - Extracting all export and table parameter values
  - De-marshalising them into appropriate client side parameter objects after each call

Server Interface Features

The Java RFC Class Library also offers classes and objects that allow you to write application programs that act as RFC server to R/3.
RFC server applications using the Java RFC can:

- Register themselves in the SAP Gateway
- Wait for incoming RFC calls
- Accept RFC function calls
- Process the data from the call
- Return results of the processing to the calling client

When handling parameters that are communicated to and from the server application, the Java RFC:

- Extracts all import and table parameter values in the RFC format
- De-marshals them into appropriate parameter objects for the application
- Packages all export and table parameter values into the appropriate RFC format
- Send them over the network

**General Feature of the java RFC Class Library**

The features the Java RFC Class Library offers for building RFC client or server applications are:

- Pluggable middleware architecture

  We refer to the underlying implementation of the Java RFC (as opposed to the interface definition) as Middleware.

  You can use a middleware type of your choice. The Java RFC class Library comes with a [CORBA](Page 10) and a [JNI](Page 12) implementation of the middleware. See [Middleware Used by Java RFC](Page 9).

- The interfaces provided are designed to hide middleware implementation details. The application can specify the middleware type to use through two properties files. You then use the properties files when working with the global [SessionManager](Page 43) object. Most of your application code can remain unchanged regardless of the middleware implementation you use. It is not recommended, however, to switch middleware while a Java RFC application is running.

- Creation of connection objects (for RFC client applications), function and function module objects (for RFC server and client applications respectively), and parameter objects (including table objects) are all done through factories. Different middleware implementers supply different factories, yet from the application’s view they all support the same set of interfaces.
Java RFC Features

- Function module objects and table/structure parameter objects can be created manually or automatically. When creating objects manually, the application constructs the metadata for the function module and its parameters. When creating objects automatically, the application does not need to construct metadata information. Java RFC obtains the metadata from R/3 at runtime (with the overhead of additional RFC calls).

- When setting or getting parameter values, all RFC data types are supported, and they can be converted to any Java native types on demand when possible. For example, for an RFC NUMC (numerical string) type field, the application can ask to get the value in the form of any of the following: String, byte, short, int, long, BigInteger, BigDecimal, float, double, or byte array, as long as the value can be converted into these forms. If the conversion is impossible, an exception is thrown (for example, if you attempt to convert a number into short when the number is larger than the maximum short value).

To improve the efficiency of transferring table data, Java RFC performs:

- Smart caching of rows, meaning that a row access will result in buffering of adjacent rows. The application can set each table’s read buffer size at any time.

- Delta management of data, meaning that the Java RFC only transports data that had changed to the R/3 system, and not the whole data set.

- Java RFC provides exception handling

- Both RFC client and RFC server applications using the Java RFC are multi-thread safe.

Additional Orbix Server Features

The following are items that are unique to the Orbix middleware:

- The Orbix server uses the SAP RFC C++ Class Library and has client side multi-threading capabilities.

- The Orbix server process uses a pool of threads to handle client requests concurrently. The number of threads in the pool is configurable from the command line.

  Orbix also allows the administrator of the Orbix server to configure a number of separate server processes to handle multiple client requests.

- The Orbix server will stay active as long as there are open connections from the client. When there are no client connections, it may timeout and exit. The timeout period is configurable from the command line. The default is infinite timeout.

- The Orbix server keeps a reference list of server objects created for each client connection. These objects will stay around until the connection is closed down. Upon close of connection, the server will automatically clean up all server side objects created on behalf of this client. No client side calls for cleanup is required.

- For performance reasons, all structure and table data are transferred over the network as byte streams.

- Marshaling and unmarshaling (between the Orbix server and the Orbix client) are done implicitly.

- Uses IONA’s Orbix 2.3c for the underlying CORBA transport layer.
There are no special features or considerations when using the JNI middleware.
Java RFC Documentation

How to Use this Help Document

This document describes how to program with the Java RFC Class Library. It is geared towards developers who are building Java applications or applets that communicate with R/3 using the SAP RFC interface.

This document assumes that are familiar with the following:

- Java Programming
- R/3 RFC protocol, including setting up the necessary infrastructure for using it
- When programming an RFC client application, you also need to be familiar with the specific RFC function(s) you wish to call

Related Reference Documentation

The full reference documentation for the various client and server interfaces and classes within the Java RFC Class Library are in a separate HTML document, which we refer to as the Java RFC HTML Reference. It was previously called the Java RFC Client HTML Reference. It is available in the Java RFC program group after installation of the Java RFC Class Library.
Release Information

The Java RFC Class Library is compatible with JDK 1.1 and JDK 1.2.

This version of Java RFC supports the features of R/3 version 2.x and higher.

The Orbix server platform must be Windows NT.

The Orbix server component requires the 2.3C version of IONA Technologies' Orbix product.
What’s New in Release 4.6A?

The code for establishing a single connection to R/3 is much simpler now: less steps are required for establishing such a connection, and there is no need to refer to a connection object in methods dealing with an RFC function module within a single connection.

To provide for this simplified code, the SessionManager object has been added to the Java RFC class library. The SessionManager handles a single connection. As part of this role it can create the five factory objects of the Java RFC in the same manner as the FactoryManager can. When creating objects, the SessionManager ensures that objects have all the information they need to establish a connection.

When using a single connection the SessionManager can therefore take the place of the FactoryManager.

However, when using multiple connections in your program you should use the FactoryManager. Using the FactoryManager to handle connections or to create objects is similar to the process of creating any connection in the previous release of the Java RFC.

Note that while you can use the SessionManager to simplify new coding, the FactoryManager and all of its methods are still available as in the previous release of the Java RFC. There is no need to change existing code using the FactoryManager.

See Also

See the topic Client Side Interface [Page 41] and all of its subtopics for the details of how to use the SessionManager.

The Advanced Topics [Page 125] section describes how to program multiple connections using the FactoryManager.
What's New in Release 4.6B?

The 4.6B version of RFC Java Class Library offers server functionality. This release of the RFC Java Class Library offers new classes and interfaces for application programmers who wish to develop Java RFC server applications that run on external personal computers. These Java RFC server application programs, with the help of this new server functionality, can implement RFC function calls in Java, and then register themselves in the SAP Gateway to wait for RFC calls from a client (most likely an R/3 system). Once an RFC call is received, the Java server application program can process the data from the RFC call and then return the results to the client.

New Objects

To allow for building applications that offer server functionality the following objects have been added:

- ServerApp
- IServerFunction
- IServerProcess
- IServerFunctionFactory

See the description of these objects in Java RFC Server Classes and Interfaces [Page 101]

In addition, the IRfcFunction interface was added as a parent interface common to IRfcModule (for client applications) and IServerFunction.
What's New in Release 4.6C?

• The Java RFC Class Library can now be used with Java Native Interface (JNI) middleware. The Java RFC Class Library now comes with an SAP implementation of JNI. This is in addition to the supplied CORBA-based middleware (Orbix).

  Both the JNI and the Orbix middleware can be used for creating either RFC client or RFC server applications.

  See the overview discussion in the Middleware Used by the Java RFC [Page 9] and in CORBA-type Middleware: Orbix [Page 10], and SAP JNI Middleware [Page 12]. Also see the details of the installation [Page 23] for each of the middleware type.

  To accommodate the ability to use different middleware, the SessionInfo object now uses two properties files [Page 45], instead of one. See the discussion in that topic and in the topic Specifying Middleware Type [Page 64].

• RFC server applications (applications that act as servers to R/3 RFC calls) can now let the Java RFC Class Library automatically create server function objects [Page 107]. This allows for a dynamic creation of parameter objects, and it frees the programmer from the task of manually creating and adding the parameter objects [Page 104]. Manual creation was the only option in the previous release.

• Server applications are now multi-thread safe when more than one client makes RFC calls to them.
Installation and Setup

Purpose
Before you can use the RFC Java Class Library, you must install the necessary components. This document describes the process you must follow to install and successfully configure this software. The RFC Java Class Library consists of the following components:

- For the Orbix middleware:
  - Java RFC Orbix server component software
  - Java RFC Orbix client component software
  - Java RFC server component software
  - IONA Technologies' Orbix object middleware and OrbixWeb products (included)
- For the JNI middleware: the Java RFC Native DLL
- README files
- Documentation:
  - This online HTML Help document
  - Java RFC HTML Reference documentation

Procedure
1. Locate the R/3 Presentation CD. You install the Java RFC components from this CD as part of the installation of SAP Automation.
   You can also download the installation program or directly install this software from the SAP Automation Web site at http://www.saplabs.com.
2. You can now install the RFC Java Class Library to work with the SAP JNI middleware implementation [Page 24], or to work with the Orbix middleware implementation [Page 25].

See Also
See the instructions for installing the Java RFC with the desired middleware implementation. Also see the discussion of Middleware Used by Java RFC [Page 9] and in CORBA-type Middleware: Orbix [Page 10], and SAP JNI Middleware [Page 12].
Installing Java RFC for JNI Middleware

Use
Installing the Java RFC to work with the JNI middleware is very simple, and it is similar to installing the Orbix client portion [Page 27].

Procedure
1. Locate the Java RFC installation software at the R/3 Presentation CD or at the SAP Automation Web site.
   You install the Java RFC components from this CD as part of the installation of SAP Automation.
2. Start the installation procedure by running the JRFC.EXE file.
3. To install the necessary Java RFC components:
   You may accept all the defaults offered by the installation program. This includes answering "No" to the question regarding the installation of the Orbix Daemon as an NT service. However, your answer in this case does not make a difference.
   Make sure that at least the JRFC Client option in the Select Component dialog is checked. You can check both the client and the server options. Note, though, that the server option installs the Orbix server portion, which you do not need.
In addition to installing the Java RFC software, you must also specify middleware type to be the SAP JNI, by using the correct values in the Java RFC properties files [Page 45]. See the topic specifying the middleware type to use with the Java RFC [Page 64].

Result
Installing the client automatically copies the necessary Java RFC files to your client computer. This includes:
- The Java RFC implementation class packages
- The Java RFC client and server packages (com.sap.rfc and com.sap.rfc.exceptions)
- The Java RFC Native DLL
Orbix Installation and Setup

The Orbix middleware implementation allows you to distribute the RFC calls in your application and it allows you to create applets that make RFC calls from Java.

When you install the Java RFC to work with the Orbix middleware implementation, you can decide whether to install the Orbix client and server portions on the same computer.

In the case of using the Java RFC for Java applets, the applet and the Orbix client must be on the same computer. The Orbix server can be anywhere.

Prerequisites

- Locate the Java RFC installation software at the R/3 Presentation CD or at the SAP Automation Web site.
  
  You install the Java RFC components from this CD as part of the installation of SAP Automation.

- Decide how you want to install the Orbix server component of the Java RFC. The Orbix server component is necessary to run programs containing the Java RFC calls. The Orbix server computer is not necessarily the computer on which the R/3 application server resides. It can be a Web server, for example. It can also be on the same computer as the client. You can install the client or the server components of Java RFC separately, or you can install both at the same time.
  
  - If you install the client and Orbix server at the same time, both the client and server components are installed on the same computer.
  
  - To install the Orbix server components on another computer, install the client and server components separately, each on their target computer.

- The Orbix server component requires the 2.3C version of IONA Technologies' Orbix product. If you have installed a version of IONA's Orbix service on your server that is earlier than 2.3C, we recommend that you back up your configuration files, and any entries related to Orbix in the Windows environment, such as in the Windows Registry. See the topic Java RFC Server Orbix Installation Processing [Page 34] to determine what you need to back up.

- Get Administrative rights to the Java RFC server and all client PCs.

- Review the README files for the RFC server and client components for any late-breaking product information.

Process Flow

1. Run the installation program. To do this, during the SAP Automation installation, select the Java RFC component. The file JRFC.EXE is copied to a directory of your choice. Use the Save As dialog to specify the directory to place this file.

2. Install the client component of Java RFC to use the Java RFC interfaces in your programs. To install the client components [Page 27], simply choose the JRFC Client option during the Java RFC installation procedure.

3. Install and configure the Orbix server component of Java RFC to run a program containing Java RFC calls. You must install and configure the server components of the Java RFC before running programs that issue requests from the Java RFC client.
Orbix Installation and Setup

The Orbix server component requires the following setup:

a. **Install the Orbix server components [Page 28]** of the Java RFC on the server computer. The server computer is not necessarily the computer on which the R/3 application server resides. It can be a Web server, for example. It can also be on the same computer as the client.

b. Restart the Orbix server computer.

c. **Start the Orbix Daemon [Page 30]** before you run any program that sends Java RFC client requests. The Orbix daemon automatically starts the necessary Orbix server processes as needed, once it receives client requests.

   If you specified that the Orbix daemon is installed as an NT service, then the Orbix daemon is started automatically at system startup. You can skip this step in this case. See the discussion of the installation of the Orbix daemon as an NT service in the topic **Installing the Orbix Server [Page 28]**.

d. **Use the Orbix Server Manager utility to configure the properties of the Java RFC server [Page 32]**.
Installing the Orbix Client

Use
Installing the client components of the Java RFC allows you to use the Java RFC interfaces in a Java program, application, or applet.

Procedure
1. Start the installation procedure by running the JRFC.EXE file.
2. To install the client components, check the JRFC Client option in the Select Component dialog.

Result
Installing the client automatically copies the necessary client files to your client computer. This includes:
- The Java RFC implementation class packages
- The Java RFC client and server packages (com.sap.rfc and com.sap.rfc.exception)
- The IONA Orbix/Web Utilities and classes
Installing the Orbix Server

Use
Installing the Orbix server component copies and sets up the required Orbix server files on the Orbix server computer.

Orbix server installation is only the first step required for Orbix server setup. See the Orbix Installation and setup [Page 25] topic for other steps you must take to complete the setup of the Orbix server computer.

Prerequisites
The Orbix server computer must use the Windows NT operating system (Windows NT 4.0 or 3.5.1)

The Java RFC Class Library requires the use of the IONA Technologies’ Orbix and OrbixWeb utilities. The current version of the Java RFC requires version 2.3C of Orbix.

Procedure
1. Start the Java RFC installation procedure by running the JRFC.EXE file.

2. To install the server components, select the JRFC Server option in the Select Component dialog during the installation procedure. You can also install the client [Page 27] on the same computer.

3. Specify a destination folder for the files of the Java RFC server components. The default for CD installations is C:\SAP\JRFC.

4. If you have an installed IONA’s Orbix service of version earlier than 2.3c, you will be asked if you wish to overwrite this version. If you have such a version of Orbix installed, we recommend that you exit the Java RFC installation program and backup Orbix-related information in Windows.

   See the topic Java RFC Orbix Server Installation Processing [Page 34] to determine what you need to back up. Restart the Java RFC installation after this, and this time allow it to overwrite the existing Orbix installation.

   IONA does not recommend having two versions of Orbix running on the same computer.

5. Next, you may have to answer whether to install the Orbix daemon as an NT service. See the following section for a discussion of how to answer this question.

6. Restart the computer. You can let the installation procedure restart your computer, or you can restart it at a later time.

Deciding Whether to Install the Orbix Daemon as an NT Service
The Orbix daemon coordinates the establishment and closing of client connections to the server. On the NT platform, it can be installed either as a stand-alone Windows application or as an NT service.
The Orbix daemon starts the Java RFC Orbix server automatically when it receives its first client connection request.

The Implications of the Orbix Daemon Setup Choices

If you install the Orbix daemon as an NT service:

- Windows launches the Orbix service automatically at system startup.
- Both the Orbix service and the Java RFC Orbix server processes run in the background. Because of this, neither the daemon nor the server window is visible. If you start the Orbix daemon as an NT service, you will not be able to monitor diagnostic messages, such as the opening and closing of client connections, or the operations of the Orbix server. If you receive a server daemon error, you must go to the Control Panel to restart the Orbix Daemon service.

If you install the daemon as a stand-alone service (by not installing the Orbix daemon as an NT service):

- You must manually start the Orbix daemon as a stand-alone Windows application before you can receive any connection requests.
- Both the Orbix service and the Java RFC server processes run in the foreground. Java RFC servers are started in a console window. Some status and diagnostic messages (for example, connection and client shutdown notifications) are displayed in the Java RFC server window.

💡

You can reverse your decision regarding how the Orbix daemon is installed [Page 31] later.

After Installation

See the topic Java RFC Server Orbix Installation Processing [Page 34] if you have an existing Orbix service installed on your server computer. You may have to accommodate your existing Orbix setup (for example, when selecting port numbers) to make RFC calls from clients.
Starting the Orbix Daemon

**Use**

The Orbix daemon must be running before your clients issue any requests to the Java RFC server. Once it is running, the Orbix daemon starts the Java RFC Orbix server after it receives the first request.

If you have specified that the Orbix daemon is installed as an NT service, then the Orbix daemon starts automatically at system startup. It runs in the background, and is not visible.

If you have installed the Orbix daemon as a stand-alone service, then you need to invoke it manually before any client request occurs.

**Procedure**

- To start the Orbix daemon in the foreground, use the Java RFC program group from the Windows Start menu:
  
  \[Start \rightarrow Programs \rightarrow Java RFC \rightarrow Start Orbix Daemon\]

- To start the Orbix daemon in the background, use the Windows Control Panel:
  
  \[Control Panel \rightarrow Services\]
  
  Select *Orbix Daemon* from the list of services, and choose *Start*.

Also, you can use the Services dialog of the Control Panel to shut down the Orbix service, or to change the startup mode of the Orbix daemon if necessary. See *Changing the Orbix Daemon Setup after Installation* [Page 31] for more information about this process.
Changing the Orbix Daemon Setup After Installation

Use

After installation of the JRFC Orbix server components, you may choose to reverse your decision regarding the installation of the Orbix daemon as an NT service.

Procedure

To reverse the setup performed by the JRFC Orbix server installation, you need to perform the following tasks:

- Install or uninstall the Orbix daemon as an NT service. Use the items from the Java RFC program group [Page 38] to do so.
- Set the Orbix daemon startup mode. If you are making the Orbix daemon an NT service, you can select either automatic or manual operation at startup.

Changing the Start Mode of the Orbix daemon

1. Use the Windows Control Panel:
   
   Control Panel ➔ Services

2. Select Orbix daemon from the list of services.

3. Choose Startup.

4. Select either the Automatic or Manual Startup Type.

You can also manually start or stop the Orbix daemon using the same Windows dialog.
Configuring the Java RFC Orbix Server

Use
The Orbix run-time component includes various utilities for managing your Orbix server. The most important one is the Orbix Server Manager utility.

You can use the Orbix Server Manager to manually configure the Java RFC Orbix server. More specifically, you can:

- Specify the maximum number of Orbix servers that can be launched
- View and specify who is authorized to invoke a Java RFC Orbix server, and who can send requests to it
- Customize the Orbix server invocation command to control server timeout and Orbix server thread pool size
- Connect to one or more hosts where the Orbix daemon is running. Note that the Orbix daemon has to be started on the host(s) for the Orbix Server Manager to be able to connect.
- Manually launch or stop the Orbix server. The Orbix server's icon is animated when it is running and still when it is stopped.

Prerequisites
The Java RFC Orbix server components must be installed [Page 28], and the Orbix daemon must be started [Page 30].

Procedure
1. Invoke the Server Manager utility from the Java RFC program group in the Start menu:
   Start → Programs → Java RFC → Server Manager.

2. Use the Server Manager to enter any custom properties for the Orbix server. The Server Manager lists the Java RFC server as RFC under your server computer name. To enter this information double-click on the RFC entry to bring up the Server Record dialog.

3. Enter any of these properties for the RFC server in the Server Record dialog:

<table>
<thead>
<tr>
<th>Tab</th>
<th>Property</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Number of Orbix Server Objects</td>
<td>Use this field to specify the maximum number of instances of the Orbix server process that can be launched for multiple client requests.</td>
</tr>
<tr>
<td></td>
<td>Orbix Server Name</td>
<td>This property has to be “RFC” for Java RFC clients to obtain proper connections.</td>
</tr>
</tbody>
</table>
### Configuring the Java RFC Orbix Server

<table>
<thead>
<tr>
<th>Rights</th>
<th>Launch and Invoke Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Launches the rights control that governs who can start the Orbix server process. Invokes the rights control that governs who can use the RFC server. It also governs who can invoke the rights control's methods. Choosing &quot;all&quot; grants rights to everyone, and is the default setting for the Orbix server. You can choose to restrict rights for specific NT users.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method(s)</th>
<th>Marker/Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>This value is set by default to &quot;*&quot;, indicating that all Java RFC requests use the launch command as specified below. Do not change this entry.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method(s)</th>
<th>Launch/Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The Orbix daemon uses this command to automatically launch the Orbix server when the Orbix server receives its first client request. You can specify a customized JRFC server launch command using this property; that is, you may specify the Orbix server's timeout and thread pool size parameters by entering the complete command in this field. Enter the command in double quotes. For example, to specify an Orbix server timeout of five seconds, enter the following (include the double quotes): &quot;jrfcserv -t 5&quot;. See the topic Starting the Java RFC Orbix Server for the syntax of the Orbix server launch command. If you specify a customized launch command, it is added to the existing launch command at the Server Record dialog. Delete any existing launch command to ensure that the command you have specified takes effect.</td>
</tr>
</tbody>
</table>

4. Save your changes by choosing OK at the Server Record dialog.
Java RFC Orbix Server Installation Processing

Purpose
This document describes the processing performed by the JRFC.EXE installation utility. In all examples, assume that you install the Orbix server component onto a directory called TARGETDIR.

Process Flow
The Orbix installation procedure:

- Checks for existing Orbix versions:
  - If Orbix is not installed on your Orbix server computer, the Java RFC installation installs the necessary components of the Orbix service.
  - If you have an Orbix service of a version earlier than 2.3C installed, you have to overwrite it to continue with the Java RFC installation.
    In both of the above cases, the Java RFC installation copies the Orbix Library/Utilities (IONA's Orbix 2.3C runtime components, OrbixWeb, and the Orbix configuration files) to the Orbix server computer.
  - If you have version 2.3C of Orbix installed, then the Java RFC installation checks for certain Orbix components. It may install some missing tools or additional helper files in the target directory you specify.

- Copies the Java RFC Orbix server executable (jrfcserver.exe) and the RFC Dynamic Link Library (librfc32.dll) to your server computer.

- Self-registers the JRFC Orbix server with the Orbix daemon.

In addition, if you are installing the Orbix utilities FOR THE FIRST TIME, the JRFC.EXE installation program:

- Changes the IT_CONFIG_PATH environment variable to point to the TARGETDIR\cfg subdirectory.

- Sets the IT_CONFIG_PATH entry under HKEY_LOCAL_MACHINE in the Windows Registry to TARGETDIR\cfg.

- If you specify that the Orbix daemon is to be installed as an NT service, then the Java RFC Orbix server installation also sets the SYSTEM\CurrentControlSet\Services\Orbix daemon\Start Windows Registry entry under HKEY_LOCAL_MACHINE to 0x2. This means that the Orbix Daemon service will automatically start at system startup time.
  If you do not want to have the service start automatically, you may change the setting from the Control Panel, or modify this registry entry to 0x3.

- Sets the following lines in the Orbix configuration file (Orbix.cfg) in TARGETDIR\cfg:
  - IT_DAEMON_PORT is set to 1571
    This is required by the Orbix client, which looks for the Orbix daemon on the Orbix server host only at this port number. This is different from the default Orbix port number 1570. If you have an existing version of Orbix, the Java RFC installation does not change this entry.
Since the Java RFC setup assumes the use of Port number 1571, if you have an existing Orbix installation that uses another port number, you must set the port number used by the client side to the one defined in your Orbix configuration. You do this by using the MiddlewareInfo object. See the `get/setOrbDaemonPort` methods of MiddlewareInfo in the Java RFC HTML Reference document.

- **IT_LOCAL_DOMAIN** is set to empty
  
  This field can be overwritten from the Java RFC client, using the MiddlewareInfo object. See the `get/setLocalDomainName` methods of MiddlewareInfo in the Java RFC HTML Reference document.
  
  If you have an existing version of Orbix, the Java RFC installation also does not modify this value.

- Adds the [SAP Java RFC program group](Page 38) to the Windows Start menu.
Starting the Java RFC Orbix Server

Use

The Orbix daemon starts a Java RFC Orbix server automatically when it receives the first client request.

By default, after the Orbix daemon registers the Java RFC Orbix server, it launches the Orbix server after receiving the first client request. The daemon does so by calling the Java RFC executable (jrfcserver.exe) without parameters.

If you wish, you can add one or both of the parameters discussed below to configure the command that the daemon uses to launch the Orbix server. To make one of these changes, modify the Server Record dialog in the Orbix Server Manager utility. See the topic Configuring the Java RFC Orbix Server [Page 32] for details.

Also, you may want to start the Java RFC Orbix server manually, to eliminate the need for the first client request to wait for the server to start up. You can also use the steps in this procedure to change these parameters.

Prerequisites

The Orbix daemon must be running [Page 30] before you can launch the Java RFC server.

Procedure

You can launch the Java RFC Orbix server manually, by running the Java RFC executable from a console window (a DOS window).

The Java RFC executable is jrfcserver.exe, and it resides in the server\bin subdirectory under the directory you specify during installation.

Java RFC Server Launch Command Parameters

The Java RFC executable accepts the following parameters:

- Orbix server timeout.
  
  This parameter specifies the timeout period between client connections. This means that if the server is idle, with no client connection active for the specified length of time, it will shut down automatically.

  The default timeout is infinite. To specify any other timeout, use the –t option with the timeout in seconds. For example, to set the timeout to 5 seconds, specify the following:

  \[ jrfcserver \ -t \ 5 \]

- Orbix server thread pool size.

  The server uses this parameter to create the specified number of threads in a pool it uses when processing multiple client requests concurrently.

  The default thread pool size is 5. To specify any other pool size, use the –p option. For example to set the thread pool size to 10, specify the following:

  \[ jrfcserver \ -p \ 10 \]

You can specify both parameters on the same command line:
jrfcserv -t 5 -p 10

See Also

See the topic Configuring the Java RFC Orbix Server [Page 32] for how to apply these parameters to a registered server.
The SAP Java RFC Program Group

Use

The Java RFC program group contains the following items:

<table>
<thead>
<tr>
<th>Java RFC Program Menu Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install Orbix daemon NT Service</td>
<td>This item is added to the Java RFC program group only if the Java RFC installation performed a full installation of the Orbix service. This item allows you to configure the Orbix daemon as an NT service at any time.</td>
</tr>
<tr>
<td>Java RFC HTML Reference</td>
<td>A detailed reference document in HTML format for the various Java RFC interfaces and classes.</td>
</tr>
<tr>
<td>JRFC Client Readme</td>
<td>This document describes the new features in this release of the JRFC Client component and contains important instructions regarding its setup and use.</td>
</tr>
<tr>
<td>JRFC Orbix Server Readme</td>
<td>This document describes the new features in this release of the JRFC Orbix Server component and contains important instructions regarding its setup and use.</td>
</tr>
<tr>
<td>Orbix Server Manager</td>
<td>This item is added to the Java RFC program group if it finds an existing Orbix installation, or you are installing the Orbix utilities for the first time. However, this Orbix program does not include some necessary tools. You can use this item to configure the Java RFC Orbix server using the Orbix Service Manager utility [Page 32].</td>
</tr>
<tr>
<td>Start Orbix daemon</td>
<td>This item is added to the Java RFC program group only if the Java RFC installation performed a full installation of the Orbix service. Use this item to start the Orbix daemon [Page 30]. This is only necessary if you have chosen to install the Orbix daemon as a stand-alone Windows service. You must do so each time you start a new Windows session, before the first Java RFC client request occurs.</td>
</tr>
<tr>
<td>Uninstall Orbix daemon NT Service</td>
<td>This item is added to the Java RFC program group only if the Java RFC installation performed a full installation of the Orbix service. This item allows you to configure the Orbix daemon as stand-alone service at any time.</td>
</tr>
</tbody>
</table>

Activities

Use the Windows Start menu: Start ➔ Programs ➔ Java RFC to choose the desired item.
RFC Java Library Programming Guide
Running Programs that Use JDK 1.2 and Java RFC

JDK 1.2 CORBA Class Compatibility with JRFC

JDK 1.2 provides an implementation for `org.omg.CORBA` classes. These class files are not compatible with the Java RFC Class Library.

**Using -Xbootclasspath to Override the Default CORBA Classes Used**

If you are using the JDK 1.2 classes with Java RFC, you need to use the java `-Xbootclasspath` option to override the Java JDK 1.2 classes with the classes of the desired middleware.

For example, to specify using the Orbix implementation of the `org.omg.CORBA` classes (which is compatible with the Java RFC and is also provided with the Java RFC kit), you must issue the following command when running your Java program:

```
java -Xbootclasspath:<JRFC-install-directory>\client\classes;<classpaths-list> <Java-class-file-name>
```

Where:

<table>
<thead>
<tr>
<th>JRFC-install-directory</th>
<th>directory where the Java RFC Class Library is installed</th>
</tr>
</thead>
<tbody>
<tr>
<td>classpaths-list</td>
<td>semicolon separated list of paths to all other classes you need to use</td>
</tr>
<tr>
<td>Java-class-file-name</td>
<td>name of your Java program class file containing main</td>
</tr>
</tbody>
</table>

**Example**

Assume that the Java RFC program is installed in `c:\sap\JRFC` and JDK 1.2.1 is installed in `c:\`. To run a program called `RfcClientMain`, issue the following command at the console:

```
java -Xbootclasspath:c:\sap\JRFC\client\classes;c:\jdk1.2.1\jre\lib\rt.jar apps\RfcClientMain
```

**Deploying Your Program to an End User**

When deploying a program that uses the JDK 1.2 and the Java RFC Class Library, you may need to provide your users with a batch file containing the above command for setting the CORBA classes.

The batch files for running the sample programs delivered with the Java RFC kit [Page 77] contain a similar commands for setting up the reference to the class files.
Interface for Building Client Applications

The Java RFC Class Library offers classes and objects that allow you to write client application programs that send RFC requests to R/3.

The exposed Java RFC client Interfaces are defined in the `com.sap.rfc` package. Exceptions are defined in the `com.sap.rfc.exception` package.

The following features apply to virtually all Java RFC classes:

- Can be cloned by calling the `clone()` method
- Can be converted to a string representation by calling the `toString()` method

The next topics discuss the various interfaces and classes and how to use them.
Java RFC Client Files and Objects

The following topics discuss the various files, classes and interfaces for building RFC client interfaces.

Note that many of these files, classes and interfaces are also used when you build RFC server applications.
The SessionManager and SessionInfo Objects

Use

The SessionManager Object

RFC function calls require a connection to R/3.

When using the Java RFC Class Library you can set up and start a connection once, and then you can use that connection for multiple calls.

The Java RFC SessionManager object handles the different aspects of the connection to R/3 for RFC calls. Once you set up the required connection information [Page 80], the SessionManager handles the session. As part of this role, for example, the SessionManager ensures that any objects created during the session contain the necessary connection properties.

The SessionManager can handle a single connection. If you wish to use multiple connections [Page 126], use the FactoryManager object [Page 128], instead.

The SessionManager is a singleton (meaning that there is only one instance of it).

The SessionInfo Object

The SessionManager uses the SessionInfo object to hold connection information.

The information required to make a connection is divided into three categories:

<table>
<thead>
<tr>
<th>Category</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middleware information</td>
<td>Description of the implementation of the middleware software you are using with the Java RFC Class Library [Page 9]. In case of the Orbix middleware, also information on the computer acting as the Object Request Broker (ORB) server</td>
</tr>
<tr>
<td>Connection information</td>
<td>Different parameters of the R/3 system to connect to, such as R/3 application server name and system details</td>
</tr>
<tr>
<td>User information</td>
<td>User logon details such as user name, password, and language</td>
</tr>
</tbody>
</table>

The Java RFC SessionInfo object contains all of the above categories of connection information.

MiddlewareInfo, ConnectInfo, and UserInfo Objects

The SessionInfo object comprises the MiddlewareInfo, ConnectInfo, and UserInfo objects. They each contain one of the categories of the information in the SessionInfo object.

<table>
<thead>
<tr>
<th>Object</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiddlewareInfo</td>
<td>Middleware information</td>
</tr>
<tr>
<td>ConnectInfo</td>
<td>Connection information</td>
</tr>
<tr>
<td>UserInfo</td>
<td>User information</td>
</tr>
</tbody>
</table>

Integration

The following diagram summarizes the relationship between the various connection-related objects.
The SessionManager and SessionInfo Objects

See Also

The Java RFC Properties Files [Page 45], Setting Up and Starting a Session [Page 80], Using the Properties Files to Set Up SessionInfo [Page 82]

Also see the Java RFC HTML Reference documentation for the details of each of the classes.
The Java RFC Properties Files

Use

Information Required by the SessionManager

The SessionManager object [Page 43] needs the following information for both RFC client and RFC server applications:

- Which middleware implementation to use with the Java RFC
- Connection information for logging onto the R/3 system when necessary

The SessionManager needs to know which middleware implementation to use in order to create the appropriate factory objects.

Connection information is essential when creating an RFC client application because your application needs to log onto the server R/3 system. Connection information is also necessary in RFC server applications when you are creating any of the function objects automatically [Page 107]. The SessionManager needs this information because it connects to the R/3 system to get function metadata.

Specifying the Required Information

You can set the values of all of these parameters programmatically, by setting the appropriate MiddlewareInfo, ConnectionInfo, and UserInfo objects.

However, the best way to define this information is through special Java properties files used by the SessionManager.

Java RFC Properties Files

The SessionManager uses two properties files, in conjunction with the SessionInfo object [Page 43] to obtain this information.

The two properties files are described in the following table.

<table>
<thead>
<tr>
<th>Properties File Name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>r3_connection.props</td>
<td>Contains information equivalent to the information in the UserInfo, ConnectionInfo, and MiddlewareInfo objects. More specifically for middleware, it specifies the type of middleware used. If Orbix is used as the middleware implementation, then it specifies the computer acting as the Object Request Broker (ORB) server. If you specify the middleware implementation to use as custom, (by specifying 4 for middleware type - see the contents of the file below) then the SessionManager looks at the second properties file (jrfc.props), for the details of the middleware.</td>
</tr>
<tr>
<td>jrfc.props</td>
<td>Specifies the files that implement the various factory interfaces. By specifying the factory interfaces you specify the type of middleware implementation to use.</td>
</tr>
</tbody>
</table>

Using the properties files allows you to define middleware implementation and connection information in advance, and independently from your program. This allows you to work with
The Java RFC Properties Files

different middleware implementations or with different R/3 systems without changing your program code.

**Location of the Properties Files**

The Java RFC installation places a sample `r3_connection.props` file and two `jrfc.props` files in the following directory:

C:\SAP\JRFC\Client\Property Files

The supplied `jrfc.props` files are one each for the Orbix and the SAP JNI middleware implementation.

The properties files should resides in:

user.home/sap/user.name/r3_connection.props

The `user.home` and the `user.name` elements of this path are the standard Java.lang.System property names for the user home directory and user name respectively.

After installation, move or create the files in the above directory.

It is easier to create your properties files by copying the provided sample files. You can then supplement and change the `r3_connection.props` to fit your needs. We especially recommend that you copy the `jrfc.props` file for the middleware implementation you wish to use, since the sample files contains all the necessary values for working with the desired middleware implementation.

**Contents of the Properties Files**

**r3_connection.props Fields**

The following table describes the fields in the `r3_connection.props` properties file.

Fields listed in bold are required fields. Most of the fields are required information for establishing a connection. Some are required middleware information. All other fields are optional. However, all of the fields in the properties file are optional in the sense that you do not have to include any or all of the fields in the properties file to begin with (You can instead set the information directly into the relevant property of SessionInfo).
The Java RFC Properties Files

<table>
<thead>
<tr>
<th>Properties File Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jrfc.middleware.type</td>
<td>int</td>
<td>Number indicating the middleware used for implementing the interface. Use 4 to specify that the middleware is defined through the jrfc.props properties file. You can use the direct values for specifying middleware type: 1 for Orbix or 3 for the Sap JNI. However, we recommend that you use the indirect definition of middleware type by using the value 4. This indirect definition of middleware type allows you to keep the r3_connections.props file as read-only, which is what we recommend that you do if you wish to preserve password information in that file (see Setting Up and Starting a Session [Page 80]).</td>
</tr>
<tr>
<td>jrfc.middleware.orbServer.name</td>
<td>str</td>
<td>Name of the computer acting as the ORB server</td>
</tr>
<tr>
<td>jrfc.middleware.orbServer.port</td>
<td>int</td>
<td>Port number of the ORB computer</td>
</tr>
<tr>
<td>jrfc.middleware.orbServer.domain</td>
<td>str</td>
<td>Network domain of the ORB computer</td>
</tr>
<tr>
<td>jrfc.middleware.orbServer.diagnosticsLevel</td>
<td>int</td>
<td>Level of ORB messages shown. Higher level shows more messages. Orbix uses the following values for message level: 0 None, 1 Simple, 2 Full</td>
</tr>
<tr>
<td>jrfc.connection.isLoadBalancing</td>
<td>str</td>
<td>If true, use load balancing, that is, connect to the least loaded, available R/3 application server. The default is to not use load balancing.</td>
</tr>
</tbody>
</table>
## The Java RFC Properties Files

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jrfc.connection.host.name</code></td>
<td>str</td>
<td>Name of the R/3 application server to connect to.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can specify the name of the server as an IP address, or in the format:</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>computername.networkname.domain name</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can prefix the computer name with router name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is a required field when you do not use load balancing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If you use load balancing, then you must specify the next three fields,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>instead.</td>
</tr>
<tr>
<td><code>jrfc.connection.group.name</code></td>
<td>str</td>
<td>Group of computers from which the message server will choose an application server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required field when using load balancing.</td>
</tr>
<tr>
<td><code>jrfc.connection.messageServer</code></td>
<td>str</td>
<td>Computer coordinating the load balancing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required field when using load balancing.</td>
</tr>
<tr>
<td><code>jrfc.connection.system.name</code></td>
<td>str</td>
<td>Name of the R/3 system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Required field when using load balancing.</td>
</tr>
<tr>
<td><code>jrfc.connection.system.number</code></td>
<td>str</td>
<td>R/3 system number</td>
</tr>
<tr>
<td><code>jrfc.connection.gateway.host</code></td>
<td>str</td>
<td>Host name of the computer on which the RFC gateway process resides, if it is different from the R/3 application server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The RFC gateway process dispatches RFC requests to R/3. If it resides on the same computer as the R/3 application server, you do not need to specify this parameter.</td>
</tr>
<tr>
<td><code>jrfc.connection.gateway.service</code></td>
<td>str</td>
<td>The name of the RFC gateway service, which is the service name entry in the SAP Services file (for example &quot;sapgw16&quot;).</td>
</tr>
<tr>
<td><code>jrfc.connection.mode</code></td>
<td>int</td>
<td>Connection mode</td>
</tr>
</tbody>
</table>
The Java RFC Properties Files

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jrfc.connection.destination</td>
<td>str</td>
<td>Destination lookup name used by saprfc.ini or sideinfo file</td>
</tr>
<tr>
<td>jrfc.connection.isCheckingAuthorization</td>
<td>boolean</td>
<td>If True, verify password upon connecting to R/3. The default is True. If not checked at connection time, the user authorization will be checked at the first RFC call that requires a valid user logon.</td>
</tr>
<tr>
<td>jrfc.user.client</td>
<td>str</td>
<td>Client number</td>
</tr>
<tr>
<td>jrfc.user.name</td>
<td>str</td>
<td>User name</td>
</tr>
<tr>
<td>jrfc.user.password</td>
<td>str</td>
<td>User password</td>
</tr>
<tr>
<td>jrfc.user.language</td>
<td>str</td>
<td>Session language</td>
</tr>
<tr>
<td>jrfc.user.codePage</td>
<td>str</td>
<td>Number representing a code page, which defines character set</td>
</tr>
</tbody>
</table>

Sample `r3_connection.props` File

The following shows an example of an `r3_connection.props` file with fictitious values:

```ini
jrfc.middleware.orbServer.name=orbMachine
jrfc.middleware.type=4
jrfc.connection.host.name=/H/204.75.155.5/H/205.215.205.15
jrfc.connection.system.name=API
jrfc.connection.system.number=0
jrfc.user.client=300
jrfc.user.name=smith
jrfc.user.password=mypassword
jrfc.user.language=E
```

Because the `r3_connection.props` file uses 4 for the middleware type, then the Java RFC looks at the `jrfc.props` file.

Supplied `jrfc.props` Properties Files

The following code shows the sample `jrfc.props` file provided with the Java RFC for using Orbix:

```ini
com.sap.rfc.IRfcConnectionFactory=com.sap.rfc.orbix.JRfcConnectionFactory
com.sap.rfc.IRfcModuleFactory=com.sap.rfc.orbix.JRfcModuleFactory
com.sap.rfc.ISimpleFactory=com.sap.rfc.util.JSimpleFactory
com.sap.rfc.IStructureFactory=com.sap.rfc.orbix.JStructureFactory
com.sap.rfc.ITableFactory=com.sap.rfc.orbix.JTableFactory
```

The following code shows the sample `jrfc.props` file provided with the Java RFC for using the SAP JNI:

```ini
com.sap.rfc.IRfcConnectionFactory=com.sap.rfc.jni.JRfcConnectionFactory
com.sap.rfc.IRfcModuleFactory=com.sap.rfc.jni.JRfcModuleFactory
com.sap.rfc.ISimpleFactory=com.sap.rfc.util.JSimpleFactory
com.sap.rfc.IStructureFactory=com.sap.rfc.jni.JStructureFactory
com.sap.rfc.ITableFactory=com.sap.rfc.jni.JTableFactory
```
Integration

When it is instantiated, the SessionManager reads the r3_connection.props properties file, if it exists. The SessionManager then sets the equivalent fields of the SessionInfo object with the information from the r3_connection.props.

If middleware type is defined as custom, then the SessionManager reads the second properties file, jrfc.props, when it needs information on the factory objects.

If the r3_connection.props properties file does not exist, then the SessionManager creates it with the information from the SessionInfo object.

If the r3_connection.props properties file does exist, the SessionManager re-writes its contents after establishing a connection. Note that the SessionManager does not write password information into the r3_connection.props file.

The SessionManager only reads information from the jrfc.props file. It does not write to it.

The following diagram summarizes the role of the SessionManager in moving information between the SessionInfo object and the Properties files.

See Also

The SessionManager and the SessionInfo Objects [Page 43], Specifying Middleware Type [Page 64], Setting Up and Starting a Session [Page 80], Using the Properties File to Set Up SessionInfo [Page 82]

Also see the Java RFC HTML Reference documentation for the details of each of the classes.
The Function Module Object (IRfcModule)

Definition
An IRfcModule object represents an RFC function module.

Use
Use the IRfcModule object to call an RFC function module. The IRfcModule object manages all of the RFC function module parameters, including import, export, and table parameters.

The IRfcModule object supports all calling capabilities as an RFC client, including transactional client calls.

Depending on the parameters of the RFC function an IRfcModule object may contain one or more ITable objects, and one or more IImpExpParam objects. For every table parameter of the RFC function, the IRfcModule object contains an ITable object. For every import and export parameter, the IRfcModule object contains an IImpExpParam object, regardless of whether it is a single field parameter or a structure parameter.

You construct an IRfcModule object from IRfcModuleFactory. You obtain an IRfcModuleFactory object from the SessionManager. The SessionManager ensures that the IRfcModuleFactory obtains all the appropriate connection information.

The following diagram shows the relationship between IRfcModule and its parameter objects.

See Also
Parameter and Field Interfaces [Page 53], Setting Up the Function Module Object [Page 86]

When using multiple connections [Page 126], use FactoryManager [Page 128] instead of SessionManager to obtain the IRfcModuleFactory.

Also see the Java RFC HTML Reference documentation for the details of each of the classes and interfaces.
The Function Module Object (IRfcModule)
Parameter and Field Interfaces

Use

A parameter of an RFC function module can be either a:

- simple parameter, which can be either an export or an import parameter
- structure parameter, which can be either an export or an import parameter
- table parameter, which is both an import and an export parameter

A simple parameter of an RFC function also represent an ABAP field. A structure parameter of an RFC function also represent an ABAP structure.

To reflect this relationship, Java RFC offers a hierarchy of interfaces and classes for handling RFC function parameters and ABAP fields.

RFC Function Module Parameter Objects

The main interfaces for RFC parameter objects are the ISimple, IStructure, and ITable interfaces. They can be constructed with the ISimpleFactory, IStructureFactory, and ITableFactory objects respectively.

<table>
<thead>
<tr>
<th>Legend</th>
<th>Interface</th>
<th>Creates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ISimple</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IStructure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ITable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ISimpleFactory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IStructureFactory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ITableFactory</td>
<td></td>
</tr>
</tbody>
</table>

The following table describes the main RFC parameter and related interfaces.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISimple</td>
<td>Main interface for a simple RFC parameter. Allows you to get or set both the value and the metadata of simple fields or parameters.</td>
</tr>
<tr>
<td>IStructure</td>
<td>Main interface for a structure RFC parameter. Allows you to get or set both the value and the metadata of structure fields or parameters.</td>
</tr>
<tr>
<td>ITable</td>
<td>Main interface for a table RFC parameter. Allows you to get or set both the value and the metadata of table parameters.</td>
</tr>
<tr>
<td>ICursor</td>
<td>A cursor to the table, which points to a row in the table. Provides navigation within the table.</td>
</tr>
<tr>
<td>IRow</td>
<td>Represents a single row in a table</td>
</tr>
</tbody>
</table>

Base Interfaces for Parameters and Fields

Java RFC provides two base interfaces for RFC function parameters and ABAP fields:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Represents</th>
<th>Methods Summary</th>
</tr>
</thead>
</table>

April 2001
IParameter and Field Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Represents</th>
<th>Methods Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>IParameter</td>
<td>Base interface for classes representing RFC parameters</td>
<td>Getting or setting parameter name and metadata information</td>
</tr>
<tr>
<td>IField</td>
<td>Base interface for classes representing ABAP fields</td>
<td>Getting or setting field name and field metadata information</td>
</tr>
</tbody>
</table>

ISimple, IStructure, and ITable interfaces ultimately derive from IParameter, since they represent the three types of RFC function parameters.

Since ISimple and IStructure objects represent both a parameter of an RFC function and an ABAP field, they also derive (indirectly) from the IField interface.

The following topics discuss the IParameter and the IField base interfaces and the interfaces that extend them.

**IParameter and Related Interfaces**

IParameter and its derived interfaces represent RFC parameters.

Using IParameter and the objects that extend it you can get or set parameter metadata and values.

The following diagram shows the hierarchy of IParameter and the objects that extend it.

```
  IParameter
    ^          ^
   |          |
  IImpExpParam  IComplexParam
    |            |
  ISimpleParam
    |
  ISimple

  ITable
    |
  ITableFactory

  IRow
    |
  ICursor

  IStructure
    |
  IStructureFactory

  Legend
    Interface
    Extends
```

The following table describes the interfaces that extend IParameter:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Represents</th>
<th>Methods Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>IParameter</td>
<td>Base interface for classes representing RFC parameters</td>
<td>Getting or setting parameter name and metadata information</td>
</tr>
</tbody>
</table>
Parameter and Field Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Represents</th>
<th>Methods Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>IImpExpParam</td>
<td>Either Import and export parameters</td>
<td>The import or export parameters of an RFC function module can be either simple or structure parameters. IImpExpParam is an empty interface, serving as the common &quot;type&quot; for simple and structure parameters.</td>
</tr>
<tr>
<td>ISimpleParam</td>
<td>Simple parameters</td>
<td>Getting or setting the SimpleInfo [Page 58] for the parameter. SimpleInfo stores the metadata information of simple parameters.</td>
</tr>
<tr>
<td>IComplexParam</td>
<td>Complex parameters, namely tables and structures</td>
<td>Getting or setting the ComplexInfo [Page 58] for the parameter. ComplexInfo stores the metadata information of both structure and table parameters.</td>
</tr>
</tbody>
</table>

IField and Related Interfaces

The IField represents an ABAP field.

Using its objects you can obtain field names or values, perform data type conversions, and retrieve field metadata information.

The following diagram shows the hierarchy of IField and the objects that extend it.

The following table describes the interfaces that extend IField:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Represents</th>
<th>Methods Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>IField</td>
<td>Base interface for classes representing ABAP fields</td>
<td>Getting or setting field name and field metadata information</td>
</tr>
</tbody>
</table>
Parameter and Field Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISimpleField</td>
<td>A single data field</td>
<td>Getting and setting field values. Data conversion functionality between RFC types and Java types. For example, the <code>getString</code> method of ISimpleField provides conversion to a string from other field types.</td>
</tr>
<tr>
<td>IComplexField</td>
<td>A structured data record</td>
<td>Obtaining individual data fields (ISimpleField) or nested fields (IField, or IComplexField). Note that nested fields, which are available in the Java RFC, are not currently supported by RFC.</td>
</tr>
</tbody>
</table>

Relationship Between the Two Hierarchies

The main interfaces for RFC parameter objects may be derived from both the parameter and the field interface hierarchies:

- ISimple, representing a single value RFC parameter, is derived from both ISimpleParam (which in turn is derived from IImpExpParam) and ISimpleField, in the sense that it is both a simple parameter and a single data field.

- IStructure, representing a structure-type RFC parameter, is derived from IImpExpParam, IComplexParam, and IComplexField. This means that IStructure, which can be an import or export parameter for RFCs, is a parameter with complex structures, and represents a structured data record.

ITable, representing RFC table parameters, is derived only from IComplexParam, meaning that it is a parameter with complex structures. The IRow interface, representing a table row is derived only from IComplexField, meaning that is a structured data record.

The following diagram combines the parameter interface hierarchy and the field interface hierarchy to show the derivation of ISimple, IStructure, and ITable.
See Also

The IParameter and IField interfaces and their related objects deal with both metadata and values of parameters and fields. See the separate discussion of Parameter and Field Metadata [Page 58] for the additional objects that store the metadata portion of the information.

See the topic Using the Client Interface to make an RFC Call [Page 66] and its subtopics for how to use these objects to work with an RFC function module.

Also see the Java RFC Client HTML Reference documentation for the details of each of the interfaces.
Parameter and Field Metadata Classes

IFieldInfo, SimpleInfo, ComplexInfo

Metadata information for fields and parameters is stored in IFieldInfo and two classes that implement it:

<table>
<thead>
<tr>
<th>Interface/Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFieldInfo</td>
<td>The base interface for classes representing the metadata for fields and parameters. IFieldInfo is an abstract interface that is implemented by the classes SimpleInfo and ComplexInfo. IFieldInfo stores only the field name, type, and length.</td>
</tr>
<tr>
<td>SimpleInfo</td>
<td>Implements IFieldInfo for simple parameters or fields. Stores metadata for a single (simple) field or parameter.</td>
</tr>
<tr>
<td>ComplexInfo</td>
<td>Stores metadata for a Structure or a Table. ComplexInfo contains a list of IFieldInfos (a combination of SimpleInfos and ComplexInfos) describing the elements of the structure. The metadata for individual fields (IFieldInfo) of a structure or a table can be obtained from ComplexInfo. Because the ComplexInfo class itself implements the IFieldInfo interface, this means that a ComplexInfo may contain nested ComplexInfos of indefinite depth.</td>
</tr>
</tbody>
</table>

Metadata Objects and the Field and Parameter Objects

The IFieldInfo and its related objects are used by both the IParameter and the IField object hierarchies [Page 53]. They provide field metadata for IField and its related objects and they provide parameter metadata for IParameter and its related objects.
For example, the IFieldInfo, SimpleInfo, and ComplexInfo objects are used by the IParameter, ISimpleParam, and IComplexParam objects respectively. An IParameter object contains one IFieldInfo object. When the IParameter object is an ISimpleParam object, the IFieldInfo it contains is in fact SimpleInfo. When the IParameter object is an IComplexParam object, the IFieldInfo it contains is ComplexInfo. The following diagram illustrates this relationship.

The same relationship exists between IFieldInfo and IFIELD, ComplexInfo and IComplexField, and SimpleInfo and ISimpleField.
Parameter and Field Metadata Classes

See Also
See the topic Using the Client Interface to make an RFC Call [Page 66] and its subtopics for how to use these objects to get metadata information for an RFC function module.

Also see the Java RFC HTML Reference documentation for the details of each of the classes and interfaces.
Table, Row, and Cursor Interfaces

ITable, IRow, and ICursor

The ITable, IRow, and ICursor interfaces are three table-related interfaces. The following diagram shows the relationship between these interfaces.

The main interface, ITable, represents RFC table parameters. Objects of the IRow interface represent rows of a certain table, and can be manipulated through the table object. ICursor is provided for table navigation.

See Also

See the topic Using the Client Interface to make an RFC Call [Page 66] and its subtopics for how to use these objects to get metadata information for an RFC function module. Especially see the topics: Setting Up the Function Module Object [Page 86], Reading the Data from Export Parameters [Page 94], and Working with Table Data [Page 96].

Also see the Java RFC HTML Reference documentation for the details of each of the interfaces.
Factory Objects

Use
Factory objects are used for obtaining other objects. The com.sap.rfc package includes the following five factory objects:

<table>
<thead>
<tr>
<th>Factory</th>
<th>Creates the Object</th>
<th>Representing</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRfcConnectionFactory</td>
<td>IRfcConnection objects</td>
<td>RFC connections</td>
</tr>
<tr>
<td>IRfcModuleFactory</td>
<td>IRfcModule objects</td>
<td>RFC function modules</td>
</tr>
<tr>
<td>ISimpleFactory</td>
<td>ISimple objects</td>
<td>single-valued RFC parameters</td>
</tr>
<tr>
<td>IStructureFactory</td>
<td>IStructure objects</td>
<td>structured RFC parameters</td>
</tr>
<tr>
<td>ITableFactory</td>
<td>ITable objects</td>
<td>RFC table parameters</td>
</tr>
<tr>
<td>IServerFunctionFactory</td>
<td>IServerFunction objects</td>
<td>RFC server function objects</td>
</tr>
</tbody>
</table>

Integration
Both the SessionManager [Page 43] and the FactoryManager [Page 128] objects can obtain the same five factory objects.

Use the SessionManager for obtaining the factory objects when using a single connection [Page 80] in client applications.

When using a single connection you do not need to use an IRfcConnection object. Therefore you do not usually need to obtain the IRfcConnectionFactory from the SessionManager.

Use the FactoryManager for obtaining the factory objects, including IRfcConnectionFactory, when using multiple connections [Page 126].
See Also
See the *Java RFC HTML Reference* documentation for the details of each of the classes and interfaces.
Specifying Middleware Type

Purpose

The SessionManager object [Page 43] needs to know which middleware implementation to use with the Java RFC, for it to create the appropriate factory objects.

The SessionManager needs this information for both RFC client and RFC server applications.

The best way to specify the middleware implementation to use, is by specifying middleware type through the Java RFC properties files [Page 45].

Process Flow

1. Create the r3_connection.props properties files. Specify one of the following values for the jrfc.middleware.type field in the r3_connection.props properties files

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Use the Orbix implementation</td>
</tr>
<tr>
<td>3</td>
<td>Use the SAP JNI implementation</td>
</tr>
<tr>
<td>4</td>
<td>Use custom middleware type: use the implementation of the factory interfaces as specified in the jrfc.props file.</td>
</tr>
</tbody>
</table>

We recommend that you always use 4 as the value of the jrfc.middleware.type field.

This value can be used for a custom implementation of the middleware, that is, for a middleware implementation that is not supplied with the Java RFC.

More importantly, when you use this value, you direct Java RFC to determine the middleware type from the jrfc.props file, rather than from the r3_connection.props file. This is useful when you want to keep the r3_connection.props file as read-only, for the sake of preserving password information file (as we recommend in Setting Up and Starting a Session [Page 80]).

2. If you use custom middleware type, then also create the jrfc.props properties file.

3. The SessionManager obtains middleware information from the r3_connection.props file. The SessionManager does so when it is instantiated. The SessionManager also sets SessionInfo (and MiddlewareInfo) with the middleware type from r3_connection.props.

4. If you specify 4 for the jrfc.middleware.type field, the SessionManager obtains factory information from the jrfc.props file, when it needs it to create factory objects.

5. The SessionManager re-write the r3_connection.props file (re-writing middleware type

If the r3_connection.props properties file does not exists, then the SessionManager creates it with the middleware type from the SessionInfo object.

The SessionManager does not write into the jrfc.props file.

The following diagram summarizes this process.
For the details of the contents of the two properties files, see The Java RFC Properties Files [Page 45].

**See Also**

The SessionManager and the SessionInfo Objects [Page 43], Setting Up and Starting a Session [Page 80], Using the Properties File to Set Up SessionInfo [Page 82]

Also see the Java RFC HTML Reference documentation for the details of each of the classes mentioned here.
Using the Client Interface to Make an RFC Call

Purpose
Programming with the Java RFC Class Library allows you to make a call to an RFC function module.

Using the client interface you set the various parameters of the function module you wish to call, and then you issue the RFC call.

Prerequisites
Specify middleware type to use with the Java RFC [Page 64].

Note that you set up the middleware type through the Java RFC properties files [Page 45]. You also use the same properties files for setting up the connection parameters. We suggest that you set up the properties files only after reading all the topics regarding setting the properties files, specifying middleware type, and setting up the connection (the first step below).

Process Flow
The following steps describe the process of making an RFC function call using a single connection.

1. You must first set up the connection and start a session [Page 80].

   An RFC call requires a connection an R/3 system. The Java RFC Class Library provides the SessionManager class [Page 43] for managing all the aspects of the connection to R/3. You set up the necessary connection information and then you let the SessionManager object handle the connection.

   a. Obtain the SessionManager object.

   b. Set up connection information in SessionInfo of the SessionManager.

      You can use the r3_connection.props properties file [Page 45] to define the various connection parameters, in which case the connection information is automatically set in SessionInfo. See Using the Properties Files to Set Up SessionInfo [Page 82].

   c. Use the Open method of the SessionManager to start a session.

2. Prepare the IRfcModule and its parameters [Page 86].

   a. Obtain an IRfcModuleFactory object.

   b. Create an IRfcModule object with all of the parameters of the RFC function module.

      You can auto-create the IRfcModule object [Page 89], or you can manually create it [Page 91].

      If you manually create the IRfcModule object, you need to create and add all of its parameters.

   c. Set the value of any required or desired import parameters.

3. Make the RFC call [Page 93].

4. Retrieve returned export and table parameter values [Page 94].

5. Close down the connection using the Close method of the SessionManager.
You can close down the connection immediately after calling the function module even before checking the value of the export parameters.

The following diagram summarizes the process of making an RFC call with the client interface.
Using the Client Interface to Make an RFC Call

Get SessionManager

Properties in Properties Files?

Y → Start Session

N → Set SessionInfo

Get IRfcModuleFactory

Create IRfcModule

Import Parameters?

Y → Call & Receive IRfcModule

N → Set Import Parameters

Export Parameters?

Y → Check Export Parameters

N → End Session
See Also

Follow the links for the various individual steps within this process to see the details and an example of each of the steps.

Also see Programming Multiple Connections [Page 126] for how to call RFC functions within different connections.
Examples
Example: Calling RFC Functions with No Parameters

The following example creates an IRfcModule for the RFC function RFC_PING, which has no parameters.

This example uses a properties file to set up the necessary connection information [Page 82].

Note that the example manually creates the IRfcModule object. Since the function has no parameters there is no work involved in defining its parameters.

```java
// Obtain an instance of the SessionManager:
SessionManager sessionMgr = SessionManager.getInstance();
// Start the session:
SessionMgr.open();
// Create the factory for constructing a module:
IRfcModuleFactory moduleFac = SessionMgr.getRfcModuleFactory();
// Create the IRfcModule object
IRfcModule rfcModule = moduleFac.CreateRfcModule("RFC_PING");
// Call the RFC_PING function
rfcModule.callReceive();
// Close the session
sessionMgr.close();
```
Example: RFC Function with an Export Parameter

The following example calls the RFC function GET_SYSTEM_NAME, which has no import parameter. It has one export parameter, namely SYSTEM_NAME, providing the name of the system the client is connected to.

This example, too, uses a properties file to set up the necessary connection information [Page 82].

```java
// Obtain an instance of the SessionManager and start a session
SessionManager sessionMgr = SessionManager.getInstance();
sessionMgr.open();
// Get a module factory object and create the module
IRfcModule rfcModule = sessionMgr.getRfcModuleFactory().autoCreateRfcModule("GET_SYSTEM_NAME");
int retCode = rfcModule.callReceive();
// Get the value of the simple export parameter
String sysName = rfcModule.getSimpleExportParam("SYSTEM_NAME").getString();
System.out.println("System name = " + sysName);
// Close the session
sessionMgr.close();
```
Example: Calling an RFC Function with Parameters

The following example uses the Java RFC client interfaces to call an RFC function called BAPI_GL_ACC_GETLIST.

We use parts of this example in the topics that discuss the steps of making an RFC call [Page 66].

RFC Function Description

The BAPI_GL_ACC_GETLIST function is the internal implementation of the GETLIST BAPI (method) for the SAP business object GeneralLedgerAccount.

We use the BAPI_GL_ACC_GETLIST function not as a demonstration of how to use the SAP BAPI technology, but rather because this RFC function offers a good combination of simple, structure and table parameters.

The BAPI_GL_ACC_GETLIST function returns the list of G/L accounts (G/L account number, short text and long text) for a specified company in a specified language.

The following table lists the parameters of the BAPI_GL_ACC_GETLIST function (note that the order of the parameters within the function is not important, since RFC uses named parameters):

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Category</th>
<th>Type</th>
<th>Internal Name</th>
<th>Mandatory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_LIST</td>
<td>Imp/Exp</td>
<td>Table</td>
<td>BAPI3006_1</td>
<td>N</td>
</tr>
<tr>
<td>COMPANYCODE</td>
<td>Import</td>
<td>Field</td>
<td>COMP_CODE</td>
<td>Y</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>Import</td>
<td>Field</td>
<td>LANGU</td>
<td>N</td>
</tr>
<tr>
<td>LANGUAGE_ISO</td>
<td>Import</td>
<td>Field</td>
<td>LANGU_ISO</td>
<td>N</td>
</tr>
<tr>
<td>RETURN</td>
<td>Export</td>
<td>Structure</td>
<td>BAPIRETURN</td>
<td>N</td>
</tr>
</tbody>
</table>

The function returns the list of G/L accounts in the table parameter ACCOUNT_LIST, which has the following columns (note that the order of the columns in the table is important when creating and adding the fields to the table):

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP_CODE</td>
<td>CHAR</td>
<td>4</td>
</tr>
<tr>
<td>GL_ACCOUNT</td>
<td>CHAR</td>
<td>10</td>
</tr>
<tr>
<td>SHORT_TEXT</td>
<td>CHAR</td>
<td>20</td>
</tr>
<tr>
<td>LONG_TEXT</td>
<td>CHAR</td>
<td>50</td>
</tr>
</tbody>
</table>

The function also returns a structure containing information on any error messages. It has the following fields (note that the order of the fields in the structure is important when creating and adding the fields to the structure):

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>CHAR</td>
<td>1</td>
</tr>
<tr>
<td>CODE</td>
<td>CHAR</td>
<td>5</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>CHAR</td>
<td>220</td>
</tr>
</tbody>
</table>
Example: Calling an RFC Function with Parameters

<table>
<thead>
<tr>
<th>LOG_NO</th>
<th>CHAR</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_MSG_NO</td>
<td>NUMC</td>
<td>6</td>
</tr>
<tr>
<td>MESSAGE_V1</td>
<td>CHAR</td>
<td>50</td>
</tr>
<tr>
<td>MESSAGE_V2</td>
<td>CHAR</td>
<td>50</td>
</tr>
<tr>
<td>MESSAGE_V3</td>
<td>CHAR</td>
<td>50</td>
</tr>
<tr>
<td>MESSAGE_V4</td>
<td>CHAR</td>
<td>50</td>
</tr>
</tbody>
</table>

The internal names of the fields of both the table and the structure are the same as the name of the parameter.

Java RFC Code for Making this RFC Function Call

The following code uses a properties file for setting connection information [Page 82]. The code creates the IRfcModule object manually [Page 91].

```java
// Obtain an instance of the session manager
SessionManager sessionMgr = SessionManager.getInstance();
// Start the session:
SessionMgr.open();

// Get a module factory object
IRfcModuleFactory moduleFactory = sessionMgr.getRfcModuleFactory();

// **** Manually Create the IRfcModule object ****
IRfcModule rfcModule = moduleFactory.createRfcModule("BAPI_GL_ACC_GETLIST");
// Obtain the factories for creating parameters
ISimpleFactory simpleFactory = sessionMgr.getSimpleFactory();
IStructureFactory structureFactory = sessionMgr.getStructureFactory();
ITableFactory tableFactory = sessionMgr.getTableFactory();

// Create the import parameter(s) of the function module
ISimple paramCompany = simpleFactory.createSimple(
    new SimpleInfo(null, IFieldInfo.RFCTYPE_CHAR, 4, 0),
    "COMPANYCODE");

ISimple paramLanguage = simpleFactory.createSimple(
    new SimpleInfo(null, IFieldInfo.RFCTYPE_CHAR, 1, 0),
    "LANGUAGE");

ISimple paramLanguageIso = simpleFactory.createSimple(
    new SimpleInfo(null, IFieldInfo.RFCTYPE_CHAR, 2, 0),
    "LANGUAGEISO");

// Add the import parameters to the function module
rfcModule.addImportParam(paramCompany);
rfcModule.addImportParam(paramLanguage);
rfcModule.addImportParam(paramLanguageIso);

// Create the table parameter(s) of the function module
IFieldInfo[] tableColumns = {
```
Example: Calling an RFC Function with Parameters

```java
new SimpleInfo("COMP_CODE", IFieldInfo.RFCTYPE_CHAR, 4, 0),
new SimpleInfo("GL_ACCOUNT", IFieldInfo.RFCTYPE_CHAR, 10, 0),
new SimpleInfo("LONG_TEXT", IFieldInfo.RFCTYPE_CHAR, 50, 0),
new SimpleInfo("SHORT_TEXT", IFieldInfo.RFCTYPE_CHAR, 20, 0),
};
ComplexInfo tableMetaData = new ComplexInfo(tableColumns, null);
ITable tableAccountList = tableFactory.createTable(tableMetaData, "ACCOUNT_LIST");
    //An alternative way of creating the table parameter,
    //which automatically creates the columns of the table:
    //tableFactory.autoCreateTable("ACCOUNT_LIST", "BAPI3006_1");

    // Add the table parameter(s) to the function module
rfcModule.addTableParam(tableAccountList);

    // Create the export parameter(s) of the function module
IFieldInfo[] returnFieldTypes = {
    new SimpleInfo("CODE", IFieldInfo.RFCTYPE_CHAR, 5, 0),
    new SimpleInfo("LOG_MSG_NO", IFieldInfo.RFCTYPE_NUM, 6, 0),
    new SimpleInfo("LOG_NO", IFieldInfo.RFCTYPE_CHAR, 20, 0),
    new SimpleInfo("MESSAGE", IFieldInfo.RFCTYPE_CHAR, 220, 0),
    new SimpleInfo("MESSAGE_V1", IFieldInfo.RFCTYPE_CHAR, 50, 0),
    new SimpleInfo("MESSAGE_V2", IFieldInfo.RFCTYPE_CHAR, 50, 0),
    new SimpleInfo("MESSAGE_V3", IFieldInfo.RFCTYPE_CHAR, 50, 0),
    new SimpleInfo("MESSAGE_V4", IFieldInfo.RFCTYPE_CHAR, 50, 0),
    new SimpleInfo("TYPE", IFieldInfo.RFCTYPE_CHAR, 1, 0),
};
ComplexInfo returnType = new ComplexInfo(returnFieldTypes, null);
IStructure paramReturn = structureFactory.createStructure(returnType, "RETURN");
    //An alternative way of creating the structure parameter,
    //which automatically creates the fields of the structure:
    //structureFactory.autoCreateStructure("RETURN", "BAPIRETURN");

    // Add the export parameter(s) to the function module
rfcModule.addExportParam(paramReturn);
    // **** End of Manually Creating the IRfcModule object ****

    // An alternative way for creating the IRfcModule object,
    //which automatically creates and adds all of its parameters:
    //IRfcModule rfcModule =
    moduleFactory.autoCreateRfcModule("BAPI_GL_ACC_GETLIST");

    // Set the values of the import parameters
paramCompany.setString("3000");
paramLanguage.setString("E");

try{
    // Call the RFC function
    int retCode = rfcModule.callReceive();

    // Print out values of various parameters
```
Example: Calling an RFC Function with Parameters

```java
System.out.println("BAPI_GL_ACC_GETLIST return code = " + retCode);
System.out.println("==============================");

for (int i = 0; i < rfcModule.getExportParamCount(); i++)
{
    System.out.print(rfcModule.getExportParam(i));
}

for (int j = 0; j < rfcModule.getTableParamCount(); j++)
{
    ITable table = rfcModule.getTableParam(j);
    System.out.println("Showing table "+j);
    System.out.print(rfcModule.getTableParam(j));
    System.out.println();
}
for (int k = 0; k < rfcModule.getImportParamCount(); k++)
{
    System.out.println("Showing import param "+k);
    System.out.print(rfcModule.getImportParam(k));
    System.out.println();
}

// Handle errors
try
{
    catch (JRfcRfcAbapException e)
    {
        e.printStackTrace();
    }
    catch (JRfcRemoteServerException e)
    {
        e.printStackTrace();
    }
    catch (JRfcRfcConnectionException e)
    {
        e.printStackTrace();
    }
}
```
Additional Java RFC Client Samples

Two sets of samples are provided with the Java RFC Client package.

Sample Directories

The following table lists the directories of the sample programs, and describes the JDK version they are compatible with:

<table>
<thead>
<tr>
<th>Directory Name</th>
<th>Compatible with</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples</td>
<td>JDK 1.1.x and JDK 1.2</td>
</tr>
<tr>
<td>Samples.jdk102</td>
<td>JDK 1.02</td>
</tr>
</tbody>
</table>

Both sets of samples perform the same tasks.

Batch Files for Running the Samples

Each set of samples (subdirectories samples and samples.jdk102) contains the following three sets of sample JRFC codes (listed by the Windows batch file that runs the program):

<table>
<thead>
<tr>
<th>Batch File Name</th>
<th>Calls the Program Containing</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFCPing</td>
<td>a simple call to RFCPING</td>
</tr>
<tr>
<td>RFCCustomerGet</td>
<td>a call to RFC_CUSTOMER_GET, getting back all customer records with the specified selection criteria. The function module’s parameters are explicitly created and added to the function object in the program. Note that this sample code uses FactoryManager even though it uses a single connection. For an example using the SessionManager, see Example: Calling an RFC Function with Parameters [Page 73].</td>
</tr>
<tr>
<td>RFCCustomerGetAutoCreate</td>
<td>a call to RFC_CUSTOMER_GET, getting back all customer records with the specified selection criteria. The function module’s parameters are automatically created when creating the function object, using the “auto-create” feature. Note that this sample code uses FactoryManager even though it uses a single connection. For an example using the SessionManager, see Auto-Creating the IRfcModule Object [Page 89].</td>
</tr>
</tbody>
</table>

Sample Code Files

The following table lists the contents of some of the important files composing the sample code:

<table>
<thead>
<tr>
<th>Batch File Name</th>
<th>File</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFCPing</td>
<td>ClientMain</td>
<td>Main program. Mainly calls other programs.</td>
</tr>
<tr>
<td></td>
<td>PingClient</td>
<td>The essence of the PING call</td>
</tr>
</tbody>
</table>
Additional Java RFC Client Samples

<table>
<thead>
<tr>
<th>RFCCustomerGet and RFCCustomerGetAutoCreate</th>
<th>ClientMain</th>
<th>Main program. Mainly calls other programs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client.java</td>
<td>Code for setting up the FactoryManager and the IRfcConnection and for starting a connection.</td>
<td></td>
</tr>
<tr>
<td>GetCustomerModuleAuto</td>
<td>Code for auto-creating the function module.</td>
<td></td>
</tr>
<tr>
<td>GetCustomerModuleManual</td>
<td>Code for manually creating the function module.</td>
<td></td>
</tr>
<tr>
<td>GetCustomerClient</td>
<td>Reads the returned table parameter data. Pay special attention to the code in the SetParamValues and GetParamValues methods, where operations on table and simple parameter fields are performed</td>
<td></td>
</tr>
</tbody>
</table>

In certain cases where there are more than one way of achieving the same goal, alternative code segments are supplied in comments. Tasks such as using a table cursor and updating a table row are demonstrated.

In all of the samples, the user is responsible for entering the following information (some defaults are supplied):

- Orbix server for RFC: the host name where the JRFC server is located.
- R/3 logon information, including R/3 application server host name, system number, client, user, password and language.
- Selection criteria: for example, the customer number and name criteria for selection from the customer table.

Compiling the Samples

Prerequisite: Setting Path

To build the samples, you first need to make sure your environment variable PATH contains a path pointing to where your Java compiler is located, and CLASSPATH contains a path pointing to JDK1.1 core packages.

To set the path in Windows:

1. Choose the Windows Control Panel, System, which invokes the System properties dialog.
2. Choose the Environment tab in the System properties dialog.
3. Set the PATH variable to include the path of the directory where your Java compiler resides. (in addition to what the PATH variable already contains)
4. If you are using JDK 1.02 or 1.1.x, you also need to add the JDK classes to the CLASSPATH variable.

Compiling

Run the compile.bat utility in one of the subdirectories as described in the following table:
### Running the Samples

The following table describes the commands for running the three applications:

<table>
<thead>
<tr>
<th>Command</th>
<th>Runs the Application that Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfcping</td>
<td>RFCPING</td>
</tr>
<tr>
<td>rfccustomerget</td>
<td>RFC_CUSTOMER_GET (using manual create)</td>
</tr>
<tr>
<td>rfccustomergetac</td>
<td>RFC_CUSTOMER_GET (using auto-create)</td>
</tr>
</tbody>
</table>

### Viewing the Sample Applets

To view the three applets, you need to modify the CODEBASE line of the applet in each of the HTML files to point to the directory where you have the JRFC Client classes files installed.

Default input values can also be supplied through the applet parameters in the HTML files.

To cleanup, run the following:

```
compile clean
```
Setting Up and Starting a Session

Purpose

If you only need a single connection to R/3, use the SessionManager object [Page 43] to handle that connection.

The SessionManager object uses system properties, the SessionInfo object, and the r3_connection.props properties file [Page 45] to obtain information it needs for the connection.

To enable the SessionManager to handle the connection, you must set up the various connection properties of the SessionInfo object.

Process Flow

1. Obtain the SessionManager object.
2. Set the connection properties of SessionInfo.
   
   You can set the information in SessionInfo using one of the following two methods:
   
   Manually setting the properties of SessionInfo [Page 85]

   Storing the information in the r3_connection.props properties file and then letting the SessionManager take care of filling the information into SessionInfo [Page 82].

   Using a properties file for connection information simplifies the code you need to write for establishing a connection.

   You can override or change any of the properties in SessionInfo that were taken from the properties file [Page 83].

   Follow the links above to see the detailed procedure and examples of using each of these methods of setting the properties of SessionInfo.

3. Use the open method of SessionManager to start a session.
4. You can now set up the IRfcModule object and its parameters, and then call the function module.
5. After calling the function module, you can close the connection.

The following diagram summarizes this process.
Setting Up and Starting a Session

Result
If you the `r3_connection.props` properties file [Page 45] exists, then the SessionManager reads its information to set SessionInfo. You can modify this information.

SessionManager starts a connection using the information in SessionInfo.

If the connection is established successfully, the SessionManager then re-writes the contents of the properties file. To enable this update, make sure that the properties file is write-enabled.

However, the SessionManager never writes password information into the properties file. If you have password information in this properties file, it will no longer exist after the first connection because the SessionManager overwrites its contents. If you wish to preserve the password in the properties file, then make the file read-only. Note that setting the properties file to read-only does not result in an exception.

If there is no `r3_connection.props` properties file, the SessionManager creates it in the default directory (`user.home/sap/user.name/`) and it fills it with the information from SessionInfo.

See Also
If you wish to use multiple connections, you use IRfcConnection objects for each of the connections, and you use the FactoryManager instead of the SessionManager to create objects. See the topic Programming Multiple Connections [Page 126] for details.
Using the Properties Files to Set Up SessionInfo

Use

To provide the SessionManager [Page 43] with the necessary information for handling the connection, you must set the properties of SessionInfo.

If you set up an r3_connection.props properties file [Page 45] with all the necessary fields (including user password), starting an R/3 session is simple. You do not need to take any action to set the SessionInfo; the SessionManager sets the necessary properties automatically, taking the information from the properties file.

Procedure

All you need to do to start a session is:

1. Create the properties file [Page 45].
   
   Since the SessionManager never writes password information into the properties file, if you keep password information in the properties file and you wish to preserve it, then make the file read-only.
   
   However, if you are going to supplement or change in your program any of the information in the properties file [Page 83], then make the file write-enabled.

2. Instantiate the SessionManager object.

3. Use the open method of SessionManager to start a session.

Result

The SessionManager copies the information in the properties file into SessionInfo, and then starts a connection using SessionInfo.

If the connection is established successfully, the SessionManager re-writes the contents of the properties file (except for the password).

Next Step

You can now use this connection to create, manage and call RFC function module(s).

Your next step is to set up the IRfcModule object and all of its parameters [Page 86].

Example

The following code uses the SessionInfo properties file. It instantiates the SessionManager and then starts a session:

```java
// Obtain an instance of the SessionManager:
SessionManager sessionMgr = SessionManager.getInstance();
// Start the session:
SessionMgr.open();
```
Adding or Changing Connection Properties Used

Use
If you use an r3_connection.props properties file [Page 45] to set connection information, you can override any of the values in the properties file by setting new values in your program.

If the information in the properties file is not complete (for example, if you do not wish to store user password in the properties file), you must update SessionInfo before you can start a connection.

Procedure
1. Obtain the SessionManager object.
   When instantiated, the SessionManager creates SessionInfo and sets it with the information from the properties file.
2. Use the getSessionInfo method of the SessionManager to obtain a copy of the information that exists in SessionInfo (which is the information from the properties file).
3. Change or add to any of the properties. Note that you are changing the copy of SessionInfo.
4. Update the original SessionInfo of the SessionManager.
5. Now you can start a session by using the open method of the SessionManager.

Result
The SessionManager starts a connection using the properties of SessionInfo.
If the connection is established successfully, the SessionManager re-write the contents of the properties file (except for password information).

Next Step
You can now use the connection to create, manage and call RFC function module(s).
Your next step is to set up the IRfcModule object and all of its parameters [Page 86].

Example
The following code example assumes that the properties file contains all the necessary connection information, except for the user password. The example overrides the username value and it adds the password. It then starts a session.

// Obtain the instance of the SessionManager:
SessionManager sessionMgr = SessionManager.getInstance();
// Read session information
SessionInfo sessionInfo = SessionMgr.getSessionInfo();
java.util.Properties props = sessionInfo.getProperties();
// Change user name and add password value
//  using the JDK1.2 setProperty method
props.setProperty(SessionInfo.PROP_KEY_USER_USER_NAME, "JoeUser");
props.setProperty(SessionInfo.PROP_KEY_USER_PASSWORD, "JoePassword");
// Set the session information using a Properties object
sessionMgr.setSessionInfo(props);
Adding or Changing Connection Properties Used

// Start the session
sessionMgr.open();

The following code example performs the same tasks, but it uses UserInfo to get and set the appropriate properties.

SessionManager sessionMgr = SessionManager.getInstance();
// Read user information using UserInfo
SessionInfo sessionInfo = SessionMgr.getSessionInfo();
UserInfo userInfo = sessionInfo.getUserInfo();
// Change user name and add password using UserInfo
userInfo.setUserName("JoeUser");
userInfo.setPassword("JoePassword");
// Set the session information from UserInfo using a SessionInfo object
sessionInfo.setUserInfo(userInfo);
sessionMgr.setSessionInfo(sessionInfo);
// Start the session
sessionMgr.open();
Manually Setting the Properties of SessionInfo

Use
If you do not use a properties file [Page 45] to set up the connection parameters, you need to set the necessary connection information by manually setting the properties of the SessionInfo object.

Procedure
1. Instantiate the SessionManager object.
2. Set the properties of SessionInfo.
3. Use the open method of the SessionManager to start a session.

Result
The SessionManager starts a connection using the properties of SessionInfo.
If the connection is established successfully, the SessionManager writes the connection information into the r3_connection.props properties file (except for password information). If the properties file does not exist, it creates it. If it exists, it rewrites its contents.

Next Step
You can now use the connection to create, manage, and call RFC function module(s).
Your next step is to set up the IRfcModule object and all of its parameters [Page 86].

Example
The following code example sets the various properties of SessionInfo. It then starts a session.

```java
// Create an instance of the SessionManager:
SessionManager sessionMgr = SessionManager.getInstance();
// Enter properties using the JDK1.2 setProperty method
Properties props = new java.util.Properties();
props.setProperty(SessionInfo.PROPERTY_MIDDLEWARE_SERVER_NAME, "ORBMachine1");
//... (set other middleware properties)
props.setProperty(SessionInfo.PROPERTY_CONNECTION_HOST_NAME, "205.215.205.15");
//... (set other connection properties)
props.setProperty(SessionInfo.PROPERTY_USER_USER_NAME, "JoeUser");
//... (set other user properties)
// Set SessionInfo with the session information
sessionMgr.setSessionInfo(props);
// Start the session
sessionManager.open();
```
Setting Up the Function Module Object

Purpose
Before making an RFC function call you must prepare the IRfcModule object representing the RFC function module with all of the parameters of the RFC function.

Prerequisites
You must first obtain a SessionManager object, set up the necessary connection information and start a session [Page 80] before you can start working with RFC function modules.

Process Flow
1. Obtain an IRfcModuleFactory by using the getRfcModuleFactory method of the SessionManager.
   The SessionManager supplies the necessary connection information to the IRfcModuleFactory for creating the IRfcModule object.

2. Use the IRfcModuleFactory to create an IRfcModule object.
   There are two ways to create an IRfcModule object:

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantage/Disadvantage of Using</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manually, using the createRfcModule</td>
<td>If you create the IRfcModule object with createRfcModule, you have to explicitly create and</td>
</tr>
<tr>
<td>method of IRfcModuleFactory [Page 91]</td>
<td>add all of the parameters of the RFC function into the IRfcModule object. You have to do</td>
</tr>
<tr>
<td></td>
<td>so for all of the import, export, and table parameters of the function.</td>
</tr>
<tr>
<td></td>
<td>Using createRfcModule saves calls to R/3 for retrieving function metadata.</td>
</tr>
<tr>
<td></td>
<td>Use createRfcModule, for example, if you know that the RFC function does not have any</td>
</tr>
<tr>
<td></td>
<td>parameters.</td>
</tr>
<tr>
<td>Automatically, using the autoCreateRfcModule</td>
<td>If you use autoCreateRfcModule, the IRfcModule object is created together with all the</td>
</tr>
<tr>
<td>method of IRfcModuleFactory [Page 89]</td>
<td>necessary metadata of all the parameters of the RFC function module. You save the work</td>
</tr>
<tr>
<td></td>
<td>required for creating and adding the parameters' metadata.</td>
</tr>
<tr>
<td></td>
<td>However, using autoCreateRfcModule is less efficient, because it may result in multiple</td>
</tr>
<tr>
<td></td>
<td>calls to the R/3 system for obtaining the metadata of the function module.</td>
</tr>
</tbody>
</table>

   See the relevant topics above for the details of how to create an IRfcModule object and all of its parameters.

3. After creating the IRfcModule object with all of its parameters you need to set the value of import parameters of IRfcModule.
   You only need to set the values of the parameters representing required import parameters of the RFC function. You may set any other import parameters.
   Remember that table parameters may be both import and export parameters, and you may need to set their values as well.
If the RFC module has no parameters, you can call the function module [Page 93] immediately after creating the IRfcModule object.

The following diagram summarizes this process.

See Also
For the details of creating an IRfcModule that has parameters see one of the following topics:

- Auto-Creating the IRfcModule Object [Page 89]
- Manually Creating the IRfcModule Object [Page 91]

For examples see the following topics:

- Example: Calling an RFC Module with no Parameters [Page 71]
Setting Up the Function Module Object

- Example: Calling an RFC Module with Parameters [Page 73]
Auto-Creating the IRfcModule Object

Use

Using `autoCreateRfcModule` creates the IRfcModule object together with all the necessary metadata of all the parameters of the RFC function module.

Using `autoCreateRfcModule` eliminates the need to explicitly create and add the metadata of all the parameters of the function.

However, using `autoCreateRfcModule` is less efficient, because it may result in multiple calls to the R/3 system for obtaining the metadata of the function module.

Prerequisites

SessionManager and all the necessary connection properties must be set up [Page 80].

Procedure

4. Obtain an IRfcModuleFactory by using the `getRfcModuleFactory` method of the SessionManager.

5. Use the `autoCreateRfcModule` method of the IRfcModuleFactory to create the IRfcModule.

6. Set the values of import parameters as necessary.

The following diagram summarizes this process:

Result

The IRfcModule object is created with all of its metadata, that is, it is created with all of the parameters of the RFC function modules defined in it.
Auto-Creating the IRfcModule Object

You can access the metadata of the various parameters of the function module by using the various get methods of the IRfcModule object.

Example

The following example creates a function module for the RFC function `BAPI_GL_ACC_GETLIST`.

The `BAPI_GL_ACC_GETLIST` function returns the list of G/L accounts (G/L account number, short text and long text) for a specified company in a specified language. It has the following parameters:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Category</th>
<th>Type</th>
<th>Internal Name</th>
<th>Mandatory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_LIST</td>
<td>Imp/Exp</td>
<td>Table</td>
<td>BAPI3006_1</td>
<td>N</td>
</tr>
<tr>
<td>COMPANYCODE</td>
<td>Import</td>
<td>Field</td>
<td>COMP_CODE</td>
<td>Y</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>Import</td>
<td>Field</td>
<td>LANGU</td>
<td>N</td>
</tr>
<tr>
<td>LANGUAGE_ISO</td>
<td>Import</td>
<td>Field</td>
<td>LANGU_ISO</td>
<td>N</td>
</tr>
<tr>
<td>RETURN</td>
<td>Export</td>
<td>Structure</td>
<td>BAPIRETURN</td>
<td>N</td>
</tr>
</tbody>
</table>

This example uses auto-create to create the function module. Compare its code with the code for creating the same function module manually [Page 73].

```java
// Create the module factory
IRfcModuleFactory moduleFac = SessionMgr.getRfcModuleFactory();
// Create the IRfcModule object
IRfcModule rfcModule = moduleFac.autoCreateRfcModule("BAPI_GL_ACC_GETLIST");
// Set the values of COMPANYCODE and LANGUAGE import parameters
rfcModule.getSimpleImportParam("COMPANYCODE").setString ("3000");
rfcModule.getSimpleImportParam("LANGUAGE").setString ("E");
// Call the function module
rfcModule.callReceive();
```
Manually Creating the IRfcModule Object

Use

Creating the IRfcModule object manually means that you initially create an empty IRfcModule object and then you create and add its parameters in a separate step.

Manually creating the IRfcModule object involves more work, but it gives you more control over the creation of the parameters of the function module.

Another reason for creating the IRfcModule object manually is when you know that the RFC function does not have any parameters. In such a case, it is not necessary to create or add any parameters to the IRfcModule object. Using auto-create may result in an unnecessary connection to the R/3 system to get the metadata of the parameters of the function, when there is none.

Prerequisites

SessionManager and all the necessary connection properties must be set up [Page 80].

Procedure

1. Obtain an IRfcModuleFactory by using the getRfcModuleFactory method of the SessionManager.
2. Create the IRfcModule object using the createRfcModule method of IRfcModuleFactory.
3. Obtain an ISimpleFactory, IStructureFactory, and ITableFactory interfaces as necessary for constructing the various types of parameters that exist in the function module.
   
   For example, if the function module contains only simple parameters (individual fields), then you only need to obtain an ISimpleFactory. If the function module also contains a structure parameter, you need to also obtain an IStructureFactory.

   You only need to obtain a single factory interface for all the parameters of the same type.
4. Perform the following steps for every parameter of the RFC function module:
   a. Create the parameter.
   b. Add the parameter to the IRfcModule object.
      
      You have to perform these two steps for all of the import, export, and table parameters of the function.

      Note that RFC function parameter are named parameters, meaning that the order of the parameters within the function is not important. Therefore it is not important which parameter you create or add first.
   c. For each table and structure parameters you must also create and add the individual fields of that table or structure. To do so, perform the following steps:
      i. Instantiate an array of SimpleInfo objects representing the fields in the table or structure.
      ii. Create an ISimpleField object for every element of that array.
      iii. Create a ComplexInfo object
      iv. Assign the array of SimpleInfo objects to the ComplexInfo object.
Manually Creating the IRfcModule Object

v. Use the ComplexInfo object to construct the ITable or the IStructure object, using the ITableFactory or the IStructureFactory respectively.

Note that the order of the fields in a structure or a table is important.

As an alternative to creating all the individual fields of a table or a structure, you can auto-create the table or the structure.

After completing this step (Step 4) you have defined the IRfcModule object and all of its metadata (that is, all of the metadata of the parameters of the RFC function modules). You can now access the metadata of the various parameters of the function module by using the various get methods of the IRfcModule object.

5. Set the value of import parameters for the function module.

The following diagram summarizes the steps for manually creating an IRfcModule and its parameters. Inside the rectangular area are tasks that are performed automatically if you use auto-create to create the IRfcModule [Page 89].

Example

The code in Example: Calling an RFC Function with Parameters [Page 73] manually creates the IRfcModule object and creates and adds all the parameters to it.


**Calling an RFC Function Module**

**Use**
Making an RFC function module call to an R/3 system.

**Prerequisites**
1. A connection must be established [Page 80].
2. The IRfcModule object must have been created along with all of its parameters.
3. All the required import parameters must be set in the IRfcModule object.
See the topic Setting Up the Function Module Object [Page 86] for the details of creating and setting the IRfcModule object with all the necessary parameters.

**Procedure**
Use the `callReceive` method or a combination of the `call` and the `receive` methods of the IRfcModule object.

**Example**
Sample code for making an RFC call (Assuming `rfcModule` is the IRfcModule object.)

```java
rfcModule.callReceive();
```

**Result**
If the RFC function call is successful, returned values are available from the export and table parameter objects upon returning from `callReceive` or from `receive`.

You can now get the values of the various export parameters [Page 94].
Reading the Data from Export Parameters

Use

After calling an RFC function you can look at the resulting export parameters by using the IRfcModule and related objects.

Procedures

The following procedures suggest methods you can use for reading the data from export parameters. They are not necessarily the only methods with which you can access this data.

Reading Simple Export Parameters

1. Use the getSimpleExportParam of the IRfcModule object to get an ISimple object for the export parameter.
2. Use one of the get methods of ISimple (Inherited from ISimpleField) to get the value of the parameter depending on the type of parameter (See ISimpleField methods, such as getChar, getInt, and so on in the Java RFC HTML Reference).

Example

The following example reads the data from the export parameter SYSTEM_NAME of the RFC function GET_SYSTEM_NAME (The complete code for creating and calling the function is in Example: RFC Function with an Export Parameter [Page 72]).

```
ISimple sysNameParam = rfcModule.getSimpleExportParam("SYSTEM_NAME");
String sysName = sysNameParam.getString();
```

Reading Structure Export Parameters

1. Use the getStructExportParam of the IRfcModule object to get an IStructure object for the export parameter.
2. Use methods of the IStructure object inherited from IComplexIField to get the individual fields of the structure. For example, you can use getSimpleField to get an ISimpleField object for each of the fields of the structure.
3. Use one of the get methods of ISimpleField to get the value of a field in the structure depending on the type of field it is.

Example

The following example reads one of the fields from the structure export parameter RFCSI_EXPORT of the RFC function RFC_SYSTEM_INFO. The structure export parameter RFCSI_EXPORT contains various fields providing information on the system the client is connected to. The RFCOPSYS field in that structure provides operating system information. It is a string of 10 characters.

```java
// Obtain an instance of the SessionManager and start a session
SessionManager sessionMgr = SessionManager.getInstance();
    sessionMgr.open();
// Get a module factory object and create the module
IRfcModule rfcModule = sessionMgr.getRfcModuleFactory().autoCreateRfcModule("RFC_SYSTEM_INFO")
```
Reading the Data from Export Parameters

```java
; int retCode = rfcModule.callReceive();
// Get the structure export parameter
IStructure sysinfoStruct = rfcModule.getStructExportParam("RFCSI_EXPORT");
//get the value of the RFCOPSYS field
String OpSys = sysinfoStruct.getSimpleField("RFCOPSYS").getString();
System.out.println("Operating System = " + opSys);
// Close the session
sessionMgr.close();
```

Reading Export Data from Table Parameters

Tables are both import and export parameters. Possible steps for reading the export data from a table parameter:

1. Use the `getTableParam` method of the IRfcModule object to get an ITable object for the parameter.
2. For each of the rows of the table get an IRow object representing a copy of the row in the table.
   
   You can do so with methods of the ITable object (for example `getRow`) or by using a cursor (using the various get methods of ICursor). You can also use the standard Java ResultSet interface to get at rows of the table. See the detailed discussion in Working with Table, Rows, and Cursors [Page 96].
3. Use the `getSimpleField` method of the IRow object (inherited from IComplexIField) to get the ISimpleField object for each of the fields of the row.
4. Use one of the `get` methods of ISimpleField to get the value of an individual field in the row.

Example

The code in Example: Calling an RFC Function with Parameters [Page 73] prints out the contents of the returned table parameter of an RFC function.

Also see the examples in Working with Table Data [Page 96].
Working with Table Data

There are two ways to work with tables:

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the standard Java ResultSet object</td>
<td>Programming with standard methods of the Java language</td>
</tr>
<tr>
<td>Using the SAP Java RFC ITable, IRow, and ICursor interfaces</td>
<td>Allows you to use multiple cursors in a single table</td>
</tr>
</tbody>
</table>

Using the ITable, IRow, and ICursor interfaces of Java RFC directly you can perform the same tasks as when using the java.sql ResultSet methods.

Note that not all of the java.sql ResultSet methods were implemented in Java RFC, because not all of the methods are applicable to working with RFC tables (See the list of excluded methods in the Java RFC HTML Reference).

However, the majority of the methods were implemented, and methods that were implemented use the Java RFC ITable, IRow, and ICursor interfaces.

Using ITable, IRow, and ICursor Directly

**ITable**

Tables are stored and accessed on a row-by-row basis.

ITable provides methods for specifying how the table is handled

ITable can create ICursor objects to navigate between rows.

ITable also includes a `getResultSet` method, which allows you to create an object implementing the standard Java (java.sql) ResultSet interface.

**IRow**

Each IRow object contains a reference back to the originating ITable object, from which the metadata information (ComplexInfo) of the fields can be obtained.

An IRow inherits all methods of IComplexField.

**ICursor**

ICursor is used for navigating the table, moving between various rows.

The ITable object can create an unlimited number of ICursor objects, each containing a reference back to the table.

If you do not specify the row to point to when creating a cursor, it points to the first row by default.

Note that cursors are not necessarily updated automatically when rows are inserted or deleted from a table. We recommend that you finish your work with all cursors before adding or deleting rows.

**Row Buffering**

Tables are managed on the client side through "smart-caching": no rows are shipped from the Orbix server to the client until the client requests a certain row. When the client requests a row,
Working with Table Data

that particular row along with the next or previous few rows are cached on the client side. More rows are cached as the client accesses rows that are out of the current cache.

With the `setFetchForward` method you can specify the direction of the fetching of rows, that is, you can specify whether the rows after the requested row are fetched, or the rows before the requested row are fetched.

The size of the read buffer (in terms of number of rows included in it) for each cache operation is configurable and accessible through the methods `setReadBufferSize` and `getReadBufferSize`. You can specify no caching by setting the buffer size to 0. You can ask for the whole table to be cached by setting the buffer size to -1. The default buffer size is -1.

**Row Updates**

The `getRow` method of `ITable` and the various get methods of `ICursor` return a clone of a row in the table, not the row itself. As a result, any modifications to the returned row have no effect on the row of the originating table. To apply the changes to the originating table, call the `updateRow` method, passing to it the modified row copy.

**Update Policies**

You can control when updates of the table on the client side are sent back to the server, by specifying one of the following update policies:

<table>
<thead>
<tr>
<th>Update Policy</th>
<th>Time of Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE_POLICY_BUFFERED</td>
<td>All updates are cached on the client side until either acceptUpdate is called or until the next RFC call.</td>
</tr>
<tr>
<td>(Default)</td>
<td></td>
</tr>
<tr>
<td>UPDATE_POLICY_IMMEDIATE</td>
<td>Any call to <code>updateRow</code> results in a call back to the server to synchronize with the update.</td>
</tr>
</tbody>
</table>

The update policy can be set and retrieved through the methods `setUpdatePolicy` and `getUpdatePolicy` respectively. The default policy is UPDATE_POLICY_BUFFERED.

**Examples Using ITable, IRow, and ICursor**

The following examples show how to deal with two fields in the table parameter CUSTOMER_T (internal name: BRFCKNA1) of the RFC function RFC_CUSTOMER_GET. The two fields we show are KUNNR (customer number) and NAME1 (first line in customer name). Handling the two fields is shown as an example of how to handle all the fields in the row of the table.

**Appending a Row to a Table**

```java
//Create the table
ITable tableFactory.autoCreateTable("CUSTOMER_T", "BRFCKNA1");
IRow theRow = theTable.createEmptyRow();
theRow.getSimpleField("KUNNR").setString(custNo);
theRow.getSimpleField("NAME1").setString(custName);
theTable.appendRow(theRow);
```

**Updating a Row**

```java
ITable theTable = ...; //Create the table
IRow rowCopy = theTable.getRow(0); //get a copy of the first row
rowCopy.getSimpleField("KUNNR").setString(custNo);
```
Working with Table Data

rowCopy.getSimpleField("NAME1").setString(custName);
theTable.updateRow(0, rowCopy); // VERY IMPORTANT STEP!!

Traversing a Table

ITable theTable = ...; //Create the table
//gets the number of columns for the table
int fieldCount = theTable.getComplexInfo().getCount();
ICursor cur = theTable.createCursor();
IRow theRow = cur.getCurrentRow();  //get a copy of the first row
while( null != theRow )
{
    //prints out all fields in this row in the form of strings
    for (int nColumn = 0; nColumn < nFieldCount; nColumn++)
    {
        System.out.print(theRow.getSimpleField(nColumn).getString());
        System.out.print("	");
    }
    System.out.println("
");
    theRow = cur.getNextRow(); //get a copy of the next row
}

Examples Using the Standard Java ResultSet Interface

Appending a Row to a Table

ITable theTable = ...; //Create the table
java.sql.ResultSet resultSet = theTable.getResultSet(false);
resultSet.afterLast();
resultSet.moveToInsertRow();
resultSet.updateString("KUNNR", custNo);
resultSet.updateString("NAME1", custName);
theTable.insertRow();

Updating a Row

ITable theTable = ...; //Create the table
java.sql.ResultSet resultSet = theTable.getResultSet(false);
resultSet.first(); //move to the first row
//this is equivalent to: resultSet.absolute(1);

resultSet.updateString("KUNNR", custNo);
resultSet.updateString("NAME1", custName);
resultSet.updateRow();

Traversing a Table

ITable theTable = ...; //Create the table
java.sql.ResultSet resultSet = theTable.getResultSet(false);
int fieldCount = resultSet.getMetaData().getColumnCount();
//gets the number of columns for the table
//upon creation, resultSet's internal cursor is positioned
//before the first row
while( resultSet.next() ) //move cursor to next row
{
    //prints out all fields in this row in the form of strings
    for (int nColumn = 1; nColumn <= nFieldCount; nColumn++)
Working with Table Data

```java
{
    System.out.print(resultSet.getString(nColumn));
    System.out.print("	");
}
System.out.print("\n");
```

See Also

See the Java RFC HTML Reference documentation for the details of each of the table-related interfaces.
Interface for Building Server Applications

The Java RFC Class Library offers classes and objects that allow you to write application programs that act as an RFC Server to R/3.

These applications can:

- Register themselves in the SAP Gateway
- Wait for incoming RFC calls
- Accept RFC function calls
- Process the data from the call
- Return results of the processing to the calling client
- Offer multi-threaded, safe operations
Java RFC Server Classes and Interfaces

The exposed Java RFC client Interfaces are defined in the `com.sap.rfc` package.

Exceptions [Page 139] are defined in the `com.sap.rfc.exception` package.

When programming a server application you use mainly the following objects:

- `ServerApp`
- `IServerFunction`
- `IServerProcess`
- Factory Objects [Page 62], especially the SessionManager and the `IServerFunctionFactory`, but you can also use the FactoryManager.
- Parameter and Field Interfaces [Page 53]

The following diagram shows the relationship between the various objects used for creating Java RFC server applications.

![Diagram showing the relationship between various objects](image)

In addition to the above interfaces and classes, your RFC server application also uses many of the interfaces and classes that are used by the RFC client applications. For example, you use the SessionManager for creating the various factory interfaces, and you use all the parameter-related objects when using the `IServerFunction`. See the discussion of these objects in the appropriate sections.
Building Java RFC Server Applications

When creating server applications with the *Java RFC Class Library* you can manually create all the parameter objects for the RFC function you are serving. As an alternative, you can let the *Java RFC Class Library* automatically create all the necessary parameter objects.

**Manual Creation of Parameter Objects**

When you create server function objects manually, you perform more steps: you must construct the parameter objects for each of the server function offered by your program.

You must perform the following steps for each server function object:

- Create all of its import parameter objects (each import parameter object can be a simple parameter or a structure parameter).
- Create all of its export parameter objects (each export parameter object can be a simple parameter or a structure parameter).
- Create all of its table parameter objects.
- If a parameter is a table or a structure, you must also create and add all of the individual field objects to the parameter object.
- Add the created parameter objects to their parent server function object.

**Automatic Creation of Parameter Objects**

Letting the *Java RFC Class Library* automatically create server function objects eliminates the tedious tasks of creating and adding the parameter objects to the server function object.

To allow for automatic creation of parameter objects, you must first define the signatures needed by the server function you intend to offer in your application (This means that you create a dummy RFC function in R/3 with the parameters of the function you intend to offer).

You do so in the R/3 system, using the R/3 Function Builder (Transaction SE37).

Once you define the signatures of your function in the R/3 system, the *Java RFC Class Library* can obtain the function's metadata for the purpose of automatic creation of your server function and its parameter objects.

Note that although it is not required to do so, you may define a dummy function in the R/3 system for a function you offer even when you manually create all function objects. The dummy function in this case can be useful for testing the RFC function on the calling R/3 system.

**Comparison of the Methods**

When using automatic creation of parameter objects, the *Java RFC Class Library* makes calls to R/3 to obtain this metadata.

Therefore, while the advantage of creating function objects automatically is in saving programming effort, it results in more calls to the R/3 system, which may be less efficient at run time.

For the details of creating a server application with one of these methods, see the topics:

- Server Programming with Manual Creation of Function Objects [Page 104]
- Server Programming with Automatic Creation of Function Objects [Page 107]
Server Programming with Manual Creation of Function Objects

Use

When creating server applications with the Java RFC Class Library you can manually create all the parameter objects for the RFC function you are offering in your application.

When you create server function objects manually you perform more steps than if you let the Java RFC Class Library automatically create those objects [Page 107]: you must construct the parameter objects for each of the server functions offered by your program.

Prerequisite

Set up the two properties files [Page 45] to define the middleware type [Page 64] to use.

Procedure

<table>
<thead>
<tr>
<th>Task Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Obtain a single instance of SessionManager. See Obtaining SessionManager [Page 110] for more details and an example.</td>
</tr>
<tr>
<td>3. Use the SessionManager to create an IServerFunctionFactory object. See Using the IServerFunctionFactory Object [Page 112] for more information.</td>
</tr>
<tr>
<td>4. Obtain instances of ISimpleFactory, IStructureFactory and ITableFactory as necessary for constructing the various types of parameters that exist in all the RFC functions that you intend to offer. For example, if any of your functions contains simple parameters (individual fields), then you need to obtain an ISimpleFactory. If any of your functions contains a structure parameter, you need to also obtain an IStructureFactory, and so on. Note that you only need to obtain a single factory interface for all the parameters of the same type throughout your program (meaning that you do not need to create the factories separately for each function). Perform the following steps for every RFC function that you intend to offer in your application:</td>
</tr>
<tr>
<td>5. Use the IServerFunctionFactory to create an empty IServerFunction object representing the RFC function. Do so by calling IServerFunctionFactory.createEmptyServerFunction(). See Using the IServerFunctionFactory Object [Page 112] for more information. Perform the following steps for every parameter of the function:</td>
</tr>
</tbody>
</table>
6. Create the parameter as either a simple, structure, or table parameter object, using the appropriate factory object (ISimpleFactory, IStructureFactory and ITableFactory respectively).
   See Using the Parameter Factory Objects [Page 113] for more information.

7. If the parameter is a structure or a table parameter, then you also have to create and add all of its individual fields. To do so, perform the following steps:
   a. Instantiate an array of SimpleInfo objects representing the fields in the table or structure.
   b. Create an ISimpleField object for every element of that array.
   c. Create a ComplexInfo object for the table or structure.
   d. Assign the array of SimpleInfo objects to the ComplexInfo object.
   e. Use this ComplexInfo object to construct the ITable or the IStructure object, using the ITableFactory or the IStructureFactory respectively.

8. Call either the addImportParam(), the addExportParam() or the addTableParam() method of IServerFunction to add the parameter objects that you have created in step (6) above to the server function object.
   See Adding Parameter Objects to IServerFunction [Page 115] for more information.

9. Create a server process class that implements the IServerProcess interface. This class implements the process() method of the IServerProcess interface.
   This process() method is where you implement the processing logic of the RFC function you intend to offer. The processing logic manipulates the import data received from an incoming call, and generates export data for returning information to the calling client.

10. Instantiate a server process object belonging to your new server process class.

11. Call IServerFunction.setServerProcess(IServerProcess) to store the reference of your server process object in your server function object.
   See Associating IServerFunction and IServerProcess [Page 117] for more information.

12. Call ServerApp.addServerFunction(IServerFunction) to add the newly created and configured IServerFunction object to the single instance of ServerApp you have obtained in step (2).
   See Adding IServerFunction to ServerApp [Page 118] for more information.

13. Call the ServerApp.setAcceptArgs method to set the RFC destination where the server application is to be registered.
   See Registering a Server Application at the SAP Gateway [Page 119] for more information.
Server Programming with Manual Creation of Function Objects

<table>
<thead>
<tr>
<th></th>
<th>Adjust the number of threads for processing incoming RFC calls. See Adjusting the Number of Server Threads [Page 121] for more information</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.</td>
<td>Call ServerApp.run() to wait for incoming RFC calls and process these calls. See Running a Server Application [Page 122] for more information.</td>
</tr>
</tbody>
</table>

**Result**

You server application is ready to accept RFC calls from an R/3 system.

**See Also**

Note that the procedure for manually creating IServerFunction object is very similar to the procedure for manually creating the IRfcModule object [Page 91] (when programming a client application with the Java RFC Class Library).
Server Programming with Automatic Creation of Function Objects

Use

When creating server applications with the Java RFC Class Library you can let the Java RFC Class Library automatically create the parameter objects for the RFC functions you offer in your application.

Letting the Java RFC Class Library automatically create server function objects eliminates the tasks of creating and adding the parameter objects to the server function objects, as done with manual creation of these parameters [Page 104].

Prerequisites

- To provide the necessary metadata of the server function to the Java RFC Class Library, you must first define the signature for the server function in the calling R/3 system (This means that you create a dummy RFC function in the calling R/3 system with the parameters of the server function).

  You do so by using the Function Builder (Transaction SE37) in the calling R/3 system.

  Once you define the signatures of your function in the R/3 system, the Java RFC Class Library can obtain the function's metadata for the purpose of automatic creation of your server function and parameter objects.

- Set up the two properties files [Page 45] to define the middleware type [Page 64] to use.

- You also need to define connection parameters (in the r3_connection.props properties file) [Page 82] for logging onto the R/3 system. The connection information is required because the Java RFC connects to the R/3 system to obtains function metadata information.

Procedure

<table>
<thead>
<tr>
<th>Task Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Obtain a single instance of SessionManager.</td>
</tr>
<tr>
<td>See Obtaining SessionManager [Page 110] for more details and an example.</td>
</tr>
<tr>
<td>See Obtaining the ServerApp Object [Page 111] for more information.</td>
</tr>
<tr>
<td>3. Use the SessionManager to create an IServerFunctionFactory object.</td>
</tr>
<tr>
<td>See Using the IServerFunctionFactory Object [Page 112] for more information.</td>
</tr>
</tbody>
</table>
Perform the following steps for every RFC function that you intend to offer in your application:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.</td>
<td>Use the <code>IServerFunctionFactory</code> to create an <code>IServerFunction</code> object representing the RFC function. Call the <code>IServerFunctionFactory.createCompleteServerFunction()</code> to create all the necessary function objects automatically. See [Using the IServerFunctionFactory Object][1] for more information.</td>
</tr>
<tr>
<td>5.</td>
<td>Create a server process class that implements the <code>IServerProcess</code> interface. This class implements the <code>process()</code> method of the <code>IServerProcess</code> interface. This <code>process()</code> method is where you implement the processing logic of the RFC function you intend to offer. The processing logic manipulates the import data received from an incoming call, and generates export data for returning information to the calling client. See [Creating a Server Process for Incoming Calls][2] for more information.</td>
</tr>
<tr>
<td>6.</td>
<td>Instantiate a server process object belonging to your new server process class.</td>
</tr>
<tr>
<td>8.</td>
<td>Call <code>ServerApp.addServerFunction(IServerFunction)</code> to add the newly created and configured <code>IServerFunction</code> object to the single instance of <code>ServerApp</code> you have obtained in step (2). See [Adding IServerFunction to ServerApp][4] for more information.</td>
</tr>
<tr>
<td>9.</td>
<td>Call the <code>ServerApp.setAcceptArgs</code> method to set the RFC destination where the server application is to be registered. See [Registering a Server Application at the SAP Gateway][5] for more information.</td>
</tr>
<tr>
<td>10.</td>
<td>Adjust the number of threads for processing incoming RFC calls. See [Adjusting the Number of Server Threads][6] for more information.</td>
</tr>
<tr>
<td>11.</td>
<td>Call <code>ServerApp.run()</code> to wait for incoming RFC calls and process these calls. See <a href="#">Running a Server Application</a> [Page 122] for more information.</td>
</tr>
</tbody>
</table>

**Result**

You server application is ready to accept RFC calls from an R/3 system.
Obtaining SessionManager

Notes on Working with SessionManager

The RFC Java Class Library ensures that there is only one instance of the SessionManager for one the application program.

Example

The following example shows the typical preparation for obtaining a SessionManager reference before using it.

```java
com.sap.rfc.SessionManager _sessionMan = null;
try {
    _sessionMan.open();
} catch (JRfcBaseRuntimeException je) {
    System.out.println("Bind to object failed.\n" + "Unexpected JRFC runtime exception:\n" + je.toString());
    return;
} catch (JRfcRfcConnectionException rce) {
    System.out.println("JRfcRfcConnectionException occurred " + "when opening a session.\n" + "Exception message is: " + rce.toString());
    return;
}
```

See Also

The SessionManager is also used when creating Java RFC client applications. For more information on using the SessionManager in client applications, see The SessionManager and the SessionInfo Objects [Page 43].
Obtaining the ServerApp Object

You can only create and use one instance of ServerApp object per application program.

Example

The following example shows the typical manner in which you can obtain a reference to a single instance of this class.

```java
com.sap.rfc.ServerApp _serverApp = null;
_serverApp = com.sap.rfc.ServerApp.getInstance();
```
Using the IServerFunctionFactory Object

The IServerFunctionFactory object is used to create IServerFunction objects.

To obtain a reference to an IServerFunctionFactory object, use the SessionManager.

Once an application program obtains a reference to an IServerFunctionFactory, the program can create either an empty or a complete server function objects (depending on whether you are using manual [Page 104] or automatic [Page 107] creation of server function objects).

Note that an empty server function object does not have any parameter objects, and it is therefore useless for processing incoming RFC calls. You must create and add its parameters before accepting any incoming RFC calls to it.

Example

The following example shows how to obtain a server function factory object to create empty server function objects.

Note that this example assumes that a reference to the Factory Manager object is ready for use.

```java
// _sessionMan is a reference to SessionManager
com.sap.rfc.IServerFunctionFactory serverFactory = null;
com.sap.rfc.IServerFunction rfcComputeTax = null;
serverFactory = _sessionMan.getServerFunctionFactory();
rfcComputeTax =
serverFactory.createEmptyServerFunction("RFC_COMPUTE_TAX");

// When using manual creation
rfcComputeTax =
serverFactory.createCompleteServerFunction("RFC_COMPUTE_TAX");

// Using automatic creation
```
Using the Parameter Factory Objects to Add and Create Parameters

After creating an empty server function object you must create and add the parameter objects for all of the parameters of the function.

To create the necessary parameter objects, you must first obtain the relevant factory objects: ISimpleFactory (for creating simple parameters), IStructureFactory (for creating structure parameters), and ITableFactory (for creating table parameters).

Example

To continue the rfcComputeTax example (in Using the IServerFunctionFactory [Page 112]), which implements the RFC function RFC_COMPUTE_TAX, the following example adds the appropriate parameter objects to the function.

This example shows how to create simple and table parameter objects.

Note that this example assumes that a reference to the SessionManager object is ready for use.

```java
// _sessionMan is a reference to SessionManager
com.sap.rfc.ISimpleFactory simpleFactory = _sessionMan.getSimpleFactory();
com.sap.rfc.IStructureFactory structureFactory = _sessionMan.getStructureFactory();
com.sap.rfc.ITableFactory tableFactory = _sessionMan.getTableFactory();
com.sap.rfc.ITable _expenseTable = null;
com.sap.rfc.ISimple _name = null;

// Note that a com.sap.rfc.SimpleInfo object is needed to create a simple parameter object
_name = _simpleFactory.createSimple(new SimpleInfo(null, IFieldInfo.RFCTYPE_CHAR, 32, 0), "NAME");

// Note that an array of com.sap.rfc.SimpleInfo objects are needed to create a com.sap.rfc.ComplexInfo object first, then the complexInfo object is used to create a table or structure parameter object
IFieldInfo[] eTableInfo = new SimpleInfo[4];
int i = 0;
eTableInfo[i++] = new SimpleInfo("EXPENSE_DESCRIPTION", IFieldInfo.RFCTYPE_CHAR, 200, 0);
eTableInfo[i++] = new SimpleInfo("EXPENSE_TYPE", IFieldInfo.RFCTYPE_CHAR, 4, 0);
eTableInfo[i++] = new SimpleInfo("DATE", IFieldInfo.RFCTYPE_DATE, 8, 0);
eTableInfo[i++] = new SimpleInfo("AMOUNT", IFieldInfo.RFCTYPE_BCD, 8, 2);
_expenseTable = _tableFactory.createTable(new ComplexInfo(eTableInfo, ""));
_expenseTable.setParameterName("EXPENSE_LIST");
```
See Also

For the detailed syntax of using the factory methods, see the *Java RFC HTML Reference* documentation.
Adding Parameter Objects to IServerFunction

Most RFC functions use parameters to hold data to be processed, and to return data resulting from processing.

An RFC server function must contain parameter objects for encapsulating the parameters of the function.

Import, Export and Table Parameters

When adding parameter objects to the IServerFunction object, you must consider the direction and type of the parameter object.

Parameters can be used to perform either import or export operations. The direction is relative to the server function. All parameters that are passed from client to server are considered import parameters. All parameters that are returned from server to client are considered export parameters.

Simple and structure parameters can be either import or export parameters. Table parameters are always bi-directional.

Example

This example uses the com.sap.rfc.IServerFunction object rfcComputeTax, created in the topic Using the IServerFunctionFactory Object [Page 112]. Also, this example uses the simple and table parameters created in the Using the Parameter Factory Objects [Page 113] example.

rfcComputeTax.addImportParam(_name);
rfcComputeTax.addTableParam(_expenseTable);
Creating a Server Process for Incoming Calls

You must create a server process class that implements the `IServerProcess` interface, and implements the method `IServerProcess.process()`.

The `process()` method contains the processing logic of the RFC server function you intend to offer. Each `process()` method reads the import and table parameter information and processes the data. Then, it uses the export and table parameters to return the results of the processing.

See the discussion of simple, structure, and table parameters [Page 53] in the RFC client section of this Help document. Also see the details of each of the objects in the Java RFC HTML Reference document.

Example

```java
public class ComputeTaxProcess implements com.sap.rfc.IServerProcess {

    public ComputeTaxProcess() {
        // constructor actions
    }

    public void process(IServerFunction serverFunction) {
        // Here you put in the code that reads import
        // and table parameter data, and processes the data
        com.sap.rfc.ISimple _name =
            serverFunction.getImportParam("NAME");
        com.sap.rfc.ITable _expensesTable =
            serverFunction.getTableParam("EXPENSE_LIST");
    }
}
```
Associating IServerFunction and IServerProcess

After creating a server process class, you need to associate an instance of this server process class with the IServerFunction object you have been constructing.

To do so, first create a new instance of the server process class, then call the method IServerFunction.setServerProcess(IServerProcess) of the IServerFunction object.

Example

The following section of code shows how this is done.

The code uses the IServerFunction object we have been constructing in the examples in Using the IServerFunctionFactory Object [Page 112] and Adding Parameter Objects to IServerFunction [Page 115]. It also uses the server process class from the example in Creating a Server Process for Incoming Calls [Page 116].

```java
com.sap.rfc.IServerProcess processComputeTax =
    new ComputeTaxProcess();
rfcComputeTax.setServerProcess(processComputeTax);
```
Adding IServerFunction to ServerApp

You must add the IServerFunction, which you have created in the previous steps, to the single instance of the ServerApp object.

Example

In the following section of code, _serverApp is a reference to the single instance of ServerApp.

The IServerFunction object was created in Using the IServerFunctionFactory Object [Page 112] and in Adding Parameter Objects to IServerFunction [Page 115].

_serverApp.addServerFunction(rfcComputeTax);
Registering a Server Application at the SAP Gateway

To receive RFC calls from an R/3 client, a server application program running on an external computer must be registered in an RFC destination at an SAP Gateway. This means that you must first define an RFC destination suitable for your needs.

Defining an RFC Destination in the Calling R/3 System

To define an RFC destination on the R/3 system that will be making client RFC calls to your server application program, use Transaction SM59. In this transaction, enter an RFC destination name; for example, MYDEST. You must select T for the Connection Type and REGISTRATION for the Activation Type. Then, enter MYDEST as the program ID. After you enter this information, you are ready to save this RFC destination definition.

Defining an RFC Destination on the Application's Computer

Destination Entry in the saprfc.ini File

The RFC destination that you defined above must be reflected in the saprfc.ini file where your server application resides.

Note that in the following example saprfc.ini file, nnn.nnn.nnn.nnn is the IP address of the computer hosting the SAP Gateway (which usually resides on the same server as the R/3 system). If you wish, you do not have to use the IP address to specify the host. You can also enter the host name of the host. Also in the example below, the entry GWSERV is set to sapgw??, where ?? is the system number of the R/3 application server.

DEST=MYDEST
TYPE=R
PROGID=mydest
GHOST=n.nn.n.nn.n.nn
GWSERV=sapgw??
RFC_TRACE=1

Location of the SAPRFC.INI Files

You must place a copy of the saprfc.ini file in one of two places:

1. In the same directory as the jrfcserv.exe, which comes with RFC Java Class Library installation.
2. In a directory of your choice. Then, you need to define an environment variable RFC_INI that points to this file.

Example:

RFC_INI=C:\myprojects\rfc\java\saprfc.ini

Referring to the Destination in Your Server Application

You specify the destination to use in your server application by using the setAcceptArgs(String[]) method of the ServerApp object.
Registering a Server Application at the SAP Gateway

You specify the destination as a string containing the destination name prefixed with "-D" (note that it must be the uppercase letter D). The destination name is the same as you defined in your saprfc.ini file and on the R/3 system. You assign this string to the first element of the argument to setAcceptArgs.

Example

The following example sets the RFC destination for the server application. It assumes that all the necessary objects and variable have been defined earlier.

```java
// _serverApp is a reference to the single instance
// of com.sap.rfc.ServerApp class
String[] acceptArgs = new String[1];
acceptArgs[0] = new String();
acceptArgs[0] = "-DMYDEST";
_serverApp.setAcceptArgs(acceptArgs);
// Use uppercase D ("-D")
// in the string in acceptArgs[0] = "-DMYDEST";
```
Adjusting the Number of Server Threads

The `ServerApp` class handles the spawning and destruction of threads used in listening to incoming RFC calls and processing those calls.

The `ServerApp.run()` method starts with only one thread (in addition to itself) for listening to incoming calls.

`ServerApp` spawns more thread as necessary when more concurrent RFC calls are received. `ServerApp` stops spawning new threads when one of the following conditions are met:

- The existing number of threads are sufficient to service the incoming calls.
- The maximum allowed number of threads has been reached.

Initially, the maximum allowed number of threads is **five**. Before invoking `ServerApp.run()`, you can adjust the maximum allowed number of threads by calling `ServerApp.increaseMaximumThreadCount(int)`. Note that the argument passed to this method is the number of threads to add to the current number (the delta), rather than the absolute number for the maximum allowed.
Running a Server Application

After building the ServerApp and IServerFunction objects, you are ready to run the server application and wait for incoming calls from the client. The steps required to run a server application program are quite simple, as are illustrated by the example below.

Example

The following section of code assumes that all the preparatory steps discussed in previous sections are taken.

```java
_serverApp is a reference to the single instance of ServerApp class.
try
{
    _serverApp.run();
} catch(JRfcRfcConnectionException rce)
{
    rce.toString();
} catch(JRfcMiddlewareException me)
{
    me.toString();
}
```
Server Sample Programs: Srfctest and SrfctestAuto

The RFC Java Class Library development kit provides two sample programs to demonstrate how to construct and use the `com.sap.rfc.IServerFunction` and `com.sap.rfc.IServerProcess` objects:

<table>
<thead>
<tr>
<th>Sample Program</th>
<th>Method of Constructing Function Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Srfctest</td>
<td>manual creation</td>
</tr>
<tr>
<td>SrfctestAuto</td>
<td>automatic creation</td>
</tr>
</tbody>
</table>

Each of these programs offers the following two RFC functions:

- **STFC_CONNECTION**
- **STFC_PERFORMANCE**.

These two functions work with the R/3 program SRFCTEST. SRFCTEST includes many options, two of which can make client calls to our sample application programs: `Connection` can call `STFC_CONNECTION` and `Performance` can call `STFC_PERFORMANCE`.

**STFC_CONNECTION**

`STFC_CONNECTION` is implemented using the class `StfcConnection`.

This function receives a simple parameter of type CHAR and echoes this parameter. It then returns another simple parameter of its own, of type CHAR.

**STFC_PERFORMANCE**

`STFC_PERFORMANCE` is implemented by the class `StfcPerformance`.

This function tests the performance of a predefined set of operations. `STFC_PERFORMANCE` receives two table parameters and returns two table parameters.

**Received Table Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITAB0332</td>
<td>H332</td>
</tr>
<tr>
<td>ITAB1000</td>
<td>H1000</td>
</tr>
</tbody>
</table>

**Returned Table Parameter**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETAB0332</td>
<td>H332</td>
</tr>
<tr>
<td>ETAB1000</td>
<td>H1000</td>
</tr>
</tbody>
</table>

The `STFC_PERFORMANCE` function receives four simple parameters of type NUMC (numeric character):

<table>
<thead>
<tr>
<th>Table Parameter</th>
<th>Expected No. of Rows From R/3</th>
</tr>
</thead>
</table>
The \textit{STFC\_PERFORMANCE} function returns a single-character code, \textit{EXITCODE}. When its value is \texttt{O} \texttt{(the letter "O")}, the function call was successful. When its value is \texttt{E}, the function encountered an error during processing. This function performs the following processing:

1. The \textit{STFC\_PERFORMANCE} function checks the expected number of rows of \textit{ITAB0332} against the received number of rows of that table parameter, and then it does the same with \textit{ITAB1000}. When there is error, it returns \texttt{E} for \textit{EXITCODE}.

2. Next, the \textit{STFC\_PERFORMANCE} function appends the expected number of table rows to \textit{ETAB0332}, as indicated by \textit{LGET0332}. In each row, a header indicates the row number. The rest of the row is filled with a numeric digit.

3. Then, the \textit{STFC\_PERFORMANCE} function appends the expected number of table rows to \textit{ETAB1000}, as indicated by \textit{LGET1000}. In each row, a header indicates the row number. The rest of the row is filled with a numeric digit.

\section*{Testing the Programs}

To run the \textit{Srfctest} or the \textit{SrfctestAuto} programs as a server application:

1. Supply an RFC destination on the command line when invoking the program (see \textit{Registering a Server Application at the SAP Gateway} [Page 119] for setting up RFC destinations).

2. Log onto the R/3 System where the RFC destination is defined.

3. Use Transaction \textit{SE38}, and enter the program name: \textit{SRFCTEST}.

4. Choose either \textit{Performance} or \textit{Connection} to test the applicable function in \textit{Srfctest} or \textit{SrfctestAuto}.

5. Enter the RFC destination used by your server application.
Advanced Topics
Programming Multiple Client Connections

The procedures for starting a session [Page 80] as well as for creating and using an IRfcModule [Page 86] in the Client Side Interface sections assumed that you use a single, default connection.

When using a single connection, use the SessionManager object [Page 43] for handling all the aspects of this connection, including creating and using an IRfcModule and its parameters.

When working with multiple connections, you should instead:

- Use the FactoryManager [Page 128], instead of the SessionManager, for creating all the factory interfaces.
- Create an IRfcConnection [Page 129] object (using the IRfcConnectionFactory) for every connection you wish to use.

See the description of these objects, and the topic Handling Multiple Connections [Page 133].
Classes and Interfaces for Multiple Connections
FactoryManager

Use
When handling multiple connections, use the FactoryManager object (instead of the SessionManager object [Page 43]) as the central connection point from the client to the server.

The FactoryManager is also available for compatibility with the previous version of the Java RFC, in which the FactoryManager was the only mechanism for creating factory interfaces.

A client application or applet is only allowed to have a single FactoryManager object.

The FactoryManager object creates and manages various factory objects [Page 62] that can produce corresponding server objects on the server and proxy objects for the client.

MiddlewareInfo
The FactoryManager instance needs the MiddlewareInfo class to produce the factory objects with the right middleware implementation.

The following diagram shows the relationship between the FactoryManager and the other objects.

```
<table>
<thead>
<tr>
<th>FactoryManager</th>
<th>MiddlewareInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ISimpleFactory</td>
<td></td>
</tr>
<tr>
<td>IStructureFactory</td>
<td></td>
</tr>
<tr>
<td>ITableFactory</td>
<td></td>
</tr>
<tr>
<td>IRfcModuleFactory</td>
<td></td>
</tr>
<tr>
<td>IRfcConnectionFactory</td>
<td></td>
</tr>
</tbody>
</table>
```

The middleware implementers need to implement the various factory interfaces for creation and management of the respective server and proxy objects.

See Also
See the Java RFC HTML Reference documentation for the details of each of the classes and interfaces.
IRfcConnection and Related Objects

Definition
An IRfcConnection object represents an R/3 connection.

Use
While you can handle a single connection with the SessionManager object [Page 43], if you wish to create and use multiple connections [Page 133], you need to create an IRfcConnection object for each connection.

Once you set up the appropriate IRfcConnection object(s) you can create and use IRfcModule objects and their parameters within the context of a particular connection.

To use IRfcModule and its parameters within a particular connection you specify the IRfcConnection object in methods that create objects that require a connection (for example, when creating the IRfcModule object).

Integration
The following diagram shows the classes and interfaces related to the IRfcConnection.

You create an IRfcConnection object with the IRfcConnectionFactory object.
You create an IRfcConnectionFactory object with the FactoryManager object [Page 128].
When creating an IRfcConnection object, you provide two objects as parameters:
IRfcConnection and Related Objects

- A UserInfo object, which contains the R/3 logon user information
- A ConnectInfo object, which contains the R/3 system information

UserInfo and ConnectInfo are the same objects as used by the SessionManager [Page 43] for managing a connection. They supply the user logon information and the parameters of the system to connect to.

To create an additional connection with different user or system parameters you simply use a different IRfcConnection object, and set its UserInfo and the ConnectInfo objects with different properties.

From an established RFC connection, you can obtain the R/3 system information by calling `getSystemInfo`. The method returns a SystemInfo object, which contains a list of name-value items that describe the R/3 system properties.

The ConnectionEvent and ConnectionListener classes are for subscribing to connection open, close, or abort events. The implementation class of the IRfcConnection interface fires ConnectionEvents whenever an action is made on the connection. The subscriber of these events should implement the ConnectionListener interface.

See Also

See the Java RFC HTML Reference documentation for the details of each of the classes and interfaces.
**IRfcModule in the Context of a Connection**

**Use**

An IRfcModule object is always created within the context of a connection. **IRfcModules you create using the procedures described earlier [Page 86]** create the IRfcModule object within the single default connection.

If you **use multiple connections [Page 133]**, you need to specify the connection as a parameter when creating the IRfcModule within a connection other than the default. This is true regardless of whether you create the IRfcModule manually or with an autoCreate.

When you call an RFC function with the IRfcModule object, the relevant connection must be open. The connection must also be open whenever you use an autoCreate method to create the IRfcModule object or any of its parameters.

**Integration**

Regardless of whether it is used within a single or multiple connections, an IRfcModule object may contain one or more ITable, and one or more IImpExpParam objects.

When creating an IRfcModule for use with multiple connections, the IRfcModule object also references an IRfcConnection object.

The IRfcConnection object is created with the IRfcConnectionFactory.

The following diagram shows the relationship between IRfcModule and the other interfaces.
IRfcModule in the Context of a Connection

See Also

See the Java RFC HTML Reference documentation for the details of each of the classes and interfaces.
Handling Multiple Connections

Use

You can use multiple connections and call different RFC functions within the context of these different connections. The different connections can be to different R/3 systems or to the same system but using different user accounts, for example.

Procedure

To use multiple connections, you create one or more IRfcConnection object(s), one for each additional connection you wish to use. You create an IRfcConnection object from the IRfcConnectionFactory object, which you obtain from FactoryManager.

You then create and call the different IRfcModules within the context of the different connections.

1. Obtain the FactoryManager instance and set it up with MiddlewareInfo.
2. Obtain an IRfcConnectionFactory with the getIRfcConnectionFactory method of the FactoryManager.
3. For every connection you wish to work with (represented by IRfcConnection):
   
   a. Set the UserInfo and ConnectInfo objects with the parameters of the connection you wish to establish.
   b. Use the IRfcConnectionFactory object to create an IRfcConnection object for this connection.
      
      When creating the IRfcConnection object, specify the UserInfo and ConnectInfo as parameters.

      You can create an empty connection by not specifying UserInfo and ConnectInfo as parameters, but then you will need to set these before opening the connection.
   c. Start the connection with the open method of the IRfcConnection object.

      (Note that you start a single connection with the open method of the SessionManager object.)

4. Create the IRfcModule object within the context of the relevant connection:
   
   a. You need to specify the connection as a parameter when creating the IRfcModule within a connection. This is true regardless of whether you create the IRfcModule manually or with an autoCreate.
   b. If you are automatically creating any of the structure or table parameters of the IRfcModule object, you must also specify the connection object as a parameter to the autoCreate method (this is because AutoCreate connects to R/3 to obtain the parameter metadata).

      Make sure you use the same connection as the one you have used for creating the IRfcModule object whose parameter you are creating.
Handling Multiple Connections

When you call the RFC function module, the IRfcModule uses the connection you have specified when creating the module.

Example

The following example uses two connections to call the RFC function GET_SYSTEM_NAME twice: once in each connection. Since it connects to two different systems, the value of the resulting SYSTEM_NAME export parameter is different after the two calls. The example uses a Properties file for both connections.

```java
/**
 * Run the function in multiple connections.
 * This example connects to two different R/3 systems and calls
 * a function module on both.
 */
public void run()
{
    // Activate the Factory Manager instance
    FactoryManager factoryMgr = FactoryManager.getSingleInstance();
    SessionInfo sessionInfo = getSessionInfo(PROPS_FILE);
    factoryMgr.setMiddlewareInfo(sessionInfo.getMiddlewareInfo());

    // Obtain the necessary factories
    IRfcModuleFactory moduleFac = factoryMgr.getRfcModuleFactory();
    IRfcConnectionFactory connectionFac = factoryMgr.getRfcConnectionFactory();

    // Run a function module using the first connection
    try
    {
        // PROPS_FILE1 is a String constant representing the properties file
        SessionInfo    sessionInfo1 = getSessionInfo(PROPS_FILE1);
        IRfcConnection connection1  =
                        connectionFac.createRfcConnection(sessionInfo1.getConnectInfo(),
                                                                                                sessionInfo1.getUserInfo());

        connection1.open();

        // Specify the connection when creating the module
        IRfcModule module1 =
                                moduleFac.autoCreateRfcModule(connection1, "GET_SYSTEM_NAME");

        int    retCode1 = module1.callReceive();
        String sysName1 =
                                module1.getSimpleExportParam("SYSTEM_NAME").getString();

        System.out.println("System name = " + sysName1);

        connection1.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```
// Run a function module using the second connection
try {
    // PROPS_FILE2 is a String constant representing the properties file
    SessionInfo sessionInfo2 = getSessionInfo(PROPS_FILE2);

    IRfcConnection connection2 =
        connectionFac.createRfcConnection(sessionInfo2.getConnectInfo(),
                                           sessionInfo2.getUserInfo());

    connection2.open();

    // Specify the connection when creating the module
    IRfcModule module2 =
        moduleFac.autoCreateRfcModule(connection2, "GET_SYSTEM_NAME");

    int retCode2 = module2.callReceive();
    String sysName2 =
        module2.getSimpleExportParam("SYSTEM_NAME").getString();

    System.out.println("System name = " + sysName2);

    connection2.close();
} catch (Exception e) {
    e.printStackTrace();
}

/**
 * Obtain a SessionInfo object initialized to the values
 * in a desired properties file.
 * @param propertiesFile name of the desired properties file
 * @return
 */
private SessionInfo getSessionInfo(String propertiesFile) {
    SessionInfo sessionInfo = new SessionInfo();

    try {
        Properties properties = new Properties();

        properties.load( new java.io.BufferedInputStream( new java.io.FileInputStream(propertiesFile)));
        sessionInfo.setProperties(properties);
    } catch (java.io.IOException e) {
        // You could initialize the sessionInfo to some safe defaults.
Handling Multiple Connections

```java
    e.printStackTrace();

    return sessionInfo;
```

See Also

If you use a single connection, you can use the SessionManager object [Page 43] to handle the connection. See all the subtopics of Using the Client Interface to Make an RFC Call [Page 66].
Setting Up the Factory Manager

Use
If you use the FactoryManager object [Page 128] to create the various factory objects (instead of using the SessionManager), then you need to set it up with the appropriate middleware properties.

To use the Orbix middleware, for example, you must first initialize the Factory Manager object with this middleware type.

Procedure
Set the middleware type in your program:
1. Create a MiddlewareInfo object specifying the middleware type (for example Orbix) and the Java RFC server host.
2. Obtain the global FactoryManager object and set the MiddlewareInfo object to it.

If you wish to use a custom middleware type, or if you wish to specify the middleware type through the jrf.props properties file, then you need to specify middleware type to be MiddlewareInfo.middlewareTypeCustom. See the Java RFC Properties Files [Page 45] topic.

Note that the FactoryManager does not use the r3_connection.props properties file.

Example
Sample code to set up the FactoryManager object with the provided CORBA/Orbix middleware:

```java
MiddlewareInfo mdInfo = new MiddlewareInfo ();
mdInfo.setMiddlewareType (MiddlewareInfo.middlewareTypeOrbix);
mdInfo.setOrbServerName(rfcHost); //pass in the hostname where the
//Java RFC server is running.
facMan = FactoryManager.getSingleInstance();
facMan.setMiddlewareInfo(mdInfo);
```
Multi-Threading in RFC Client Applications

The Java RFC client packages are thread safe in the following sense:

- The global FactoryManager object is synchronized, meaning that multiple threads can concurrently invoke FactoryManager methods.
- The SessionManager object is synchronized.
- The MiddlewareInfo, UserInfo, ConnectInfo and the IRfcConnection’s implementation classes support multi-threaded subscriptions to their property change events.
- The IRfcConnection’s implementation class support multi-threaded subscriptions to its connection events.
- No class in the Java RFC implementation package has static variables. This means that multi-threaded accesses to any of the classes through the exposed interfaces are safe.

Note that the Java RFC server is also thread safe.

It is your responsibility, however, to create and handle threads in your application program, and to synchronize individual objects if there are potentially multiple threads accessing them concurrently.

For example, if your application contains an IRfcModule object, and there are multiple threads that can add or set parameter values and invoke RFC calls through it, then this object along with its parameter objects need to be synchronized by your application. Only one function call is allowed to occur in a specific connection at any given time.
Handling Exceptions and Errors

Only one error is defined for the Java RFC: JRfcRemoteOutOfMemoryError. It is thrown when the server process runs out of memory.

The Java RFC exceptions are divided into two main categories:

<table>
<thead>
<tr>
<th>Exception Category</th>
<th>Derived from</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions (JRfcBaseException)</td>
<td>java.lang.Exception</td>
<td>The base for all exceptions that need to be caught and processed by the client.</td>
</tr>
<tr>
<td>Runtime exceptions (JRfcBaseRuntimeException)</td>
<td>java.lang.RuntimeException</td>
<td>Exceptions in this category do not need to be declared or handled to allow for compilation</td>
</tr>
</tbody>
</table>

JRFC Exceptions

The following diagram shows the inheritance relationships between exception classes:

![Inheritance Diagram]

The JRfcBaseException class is the base class for all Java RFC non-runtime exceptions. Only one type of exception is directly derived from the base: JRfcIOException, from which
Handling Exceptions and Errors

JRfcRemoteException is derived. The JRfcRemoteException is the root exception class for all exceptions that can be thrown during Orbix server operations.

Among JRfcRemoteException’s children, JRfcRemoteAutoCreateException is thrown when an exception happens at the Orbix server side during an “auto-create” operation, either for creating a function module and its parameter objects, or for creating a structure or table object along with its metadata. The JRfcRemoteServerException is thrown when a general exception occurred at the Orbix server side. The JRfcRfcBaseException serves as the base for all RFC related exceptions.

Among JRfcRfcBaseException’s children, the JRfcRfcAbapException is thrown when the ABAP function module throws an exception string, such as "RECORD_NOT_FOUND". The JRfcRfcConnectionException is thrown when an error occurs during an RFC communication. The JRfcRfcParameterException is thrown when the provided RFC parameters do not match the called RFC interface, such as missing a mandatory parameter.

**JRFC Runtime Exceptions**

The following diagram shows the inheritance relationships between runtime exception classes:
The JRfcBaseRuntimeException class is the base class for all Java RFC runtime exceptions. It is derived from the java.lang.RuntimeException class, meaning that the Java RFC runtime exceptions do not need to be declared or caught for the code to compile.

JRfcBaseRuntimeException stands for all exceptions that can be thrown from any object during the normal operation of the application or applet.

You determine at what granularity to handle exceptions: you may handle the more specific exceptions (using JRfcSimpleDataCastException for example), or you may decide to handle the general-purpose errors, using the base or the base runtime exceptions.
Handling Exceptions and Errors

See the *Java RFC HTML Reference* documentation for each of the classes for details.

**Example**

Sample code for handling one specific exception, while handling all others with the base exception:

```java
try {
    int rc = module.callReceive();
} catch (JRfcRfcConnectionException re) //catch specific RFC errors
    { System.out.println("Unexpected communication error while calling RFC:\n" + re.toString());
      return;
    }

```

Sample code for handling several levels of JRFC exceptions:

```java
try {
    int retCode = rfcModule.callReceive();
}
catch (JRfcRfcAbapException e)
    { e.printStackTrace();
    }
catch (JRfcRemoteAutoCreateException e)
    { e.printStackTrace();
    }
catch (JRfcRemoteServerException e)
    { e.printStackTrace();
    }
catch (JRfcRfcConnectionException e)
    { e.printStackTrace();
    }
```

**See Also**

See the *Java RFC HTML Reference* documentation for the details of each of the exceptions.