HP NonStop SQL/MX Release 3.2.1 Guide to Stored Procedures in Java

HP Part Number: 691166-003
Published: June 2013

Printed in the US

Contents

About This Manual

This manual describes how to develop, deploy, and manage stored procedures in Java (SPJs) in NonStop SQL/MX. NonStop SQL/MX is a relational database management system based on the ANSI/ISO/IEC 9075:1999 SQL standard, commonly referred to as SQL:1999, for the HP NonStop server.

Throughout this manual, references to NonStop SQL/MX Release 2.x indicate SQL/MX Release 2.0, 2.1, 2.2, and subsequent releases until otherwise indicated in a replacement publication.

Supported Release Version Updates (RVUs)

This publication supports J06.14 and all subsequent J-series RVUs and H06.25 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Audience

This manual is intended for application programmers and database administrators who plan to implement SPJs within an SQL/MX database. The reader should know:

- The Java programming language
- JDBC and the HP NonStop JDBC Driver for SQL/MX (JDBC/MX)
- Structured query language (SQL) and SQL/MX terms and concepts

Although not required, it helps to be familiar with the part of the ANSI SQL/Foundation standard called SQL/JRT (Java Routines and Types).

Related Documentation

This manual is part of the HP NonStop SQL/MX library of manuals. The following table describes the list of manuals:

Introductory Guides

SQL/MX Comparison Guide for SQL/MP Users	Describes SQL differences between NonStop SQL/MP and NonStop SQL/MX.
SQL/MX Quick Start	Describes basic techniques for using SQL in the SQL/MX conversational interface (MXCI). Includes information about installing the sample database.

Reference Manuals

SQL/MX Reference Manual	Describes the syntax of SQL/MX statements, MXCI commands, functions, and other SQL/MX language elements.
SQL/MX Messages Manual	Describes SQL/MX messages.
SQL/MX Glossary	Defines SQL/MX terminology.

Installation Guides

SQL/MX Installation and Upgrade Guide	Describes how to plan for, install, create, and upgrade a SQL/MX database.
SQL/MX Management Manual	Describes how to manage a SQL/MX database.
NSM/web Installation Guide	Describes how to install NSM/web and troubleshoot NSM/web installations.

Connectivity Manuals

SQL/MX Connectivity Service Manual	Describes how to install and manage the HP NonStop SQL/MX Connectivity Service (MXCS), which enables applications developed for the Microsoft Open Database Connectivity (ODBC) application programming interface (API) and other connectivity APIs to use NonStop SQL/MX.
SQL/MX Connectivity Service Administrative Command Reference	Describes the SQL/MX administrative command library (MACL) available with the SQL/MX conversational interface (MXCI).
ODBC/MX Driver for Windows	Describes how to install and configure HP NonStop ODBC/MX for Microsoft Windows, which enables applications developed for the ODBC API to use NonStop SQL/MX.

Migration Guides

NonStop NS-Series Database Migration Guide	Describes how to migrate NonStop SQL/MX, NonStop SQL/MP, and Enscribe databases and applications to HP Integrity NonStop NS-series systems.

Data Management Guides

SQL/MX Data Mining Guide	Describes the SQL/MX data structures and operations to carry out the knowledge-discovery process.
SQL/MX Report Writer Guide	Describes how to produce formatted reports using data from an SQL/MX database.
DataLoader/MX Reference Manual	Describes the features and functions of the DataLoader/MX product, a tool to load SQL/MX databases.

Application Development Guides

SQL/MX Programming Manual for C and COBOL	Describes how to embed SQL/MX statements in ANSI C and COBOL programs.
SQL/MX Query Guide	Describes how to understand query execution plans and write optimal queries for an SQL/MX database.
SQL/MX Queuing and Publish/Subscribe Services	Describes how NonStop SQL/MX integrates transactional queuing and publish/subscribe services into its database infrastructure.
SQL/MX Guide to Stored Procedures in Java	Describes how to use stored procedures that are written in Java within NonStop SQL/MX.

Online Help

Reference Help	Overview and reference entries from the *SQL/MX Reference Manual*.
Messages Help	Individual messages grouped by source from the *SQL/MX Messages Manual*.
Glossary Help	Terms and definitions from the *SQL/MX Glossary*.
NSM/web Help	Context-sensitive help topics that describe how to use the NSM/web management tool.
Visual Query Planner Help	Context-sensitive help topics that describe how to use the Visual Query Planner graphical user interface.
SQL/MX Database Manager Help	Contents and reference entries from the SQL/MX Database Manager User Guide.

The NSM/web, SQL/MX Database Manager, and Visual Query Planner help systems are accessible from their respective applications. You can download the Reference, Messages, and Glossary online help from the HP Software Depot at http://www.software.hp.com. For more information about downloading the online help, see the *SQL/MX Release 3.2 Installation and Upgrade Guide*.

These manuals are part of the SQL/MP library of manuals and are essential references for information about SQL/MP Data Definition Language (DDL) and SQL/MP installation and management:

Related SQL/MP Manuals

SQL/MP Reference Manual	Describes the SQL/MP language elements, expressions, predicates, functions, and statements.
SQL/MP Installation and Management Guide	Describes how to plan, install, create, and manage an SQL/MP database. Describes installation and management commands and SQL/MP catalogs and files.

This documentation is helpful for understanding the concepts and terminology of this manual:

Other Related Documentation

NonStop Server for Java Programmer's Reference	Describes the HP NonStop Server for Java, a Java environment that supports programs for the enterprise server.
JDBC Driver for SQL/MX Programmer's Reference	Describes the JDBC Driver for NonStop SQL/MX (JDBC/MX), a type 2 driver that enables applications developed for the JDBC application programming interface (API) to use NonStop SQL/MX.
International Standard Database Language SQL—Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT) specification	Describes SQL/JRT (Java Routines and Types), which extends the ANSI SQL/Foundation standard.
Glossary of Java Technology-Related Terms	Describes Java terminology at http://java.sun.com/docs/glossary.html.

New and Changed Information

Changes to the 691166-003 manual:

- Updated the section "SQL/MX UDR Server Process" (page 25).
- Updated the section "Multithreading in an SPJ Environment" (page 27).
- Removed the 'Relink' column from Table 1 (page 32) and the para below the table.
- Updated the reference to the latest manual in "Java Library Path for the JDBC/MX Native Library" (page 35).
- Reference to connection pooling enabled by default is removed as it is no longer default.

Changes to the 691166-002 manual:

- Updated examples in the following sections:
  - "How Do I Use SPJs?" (page 18)
  - "JDBC/MX-Based Java Method" (page 54)
  - "Object Name Qualification Before NonStop SQL/MX Release 2.1.1" (page 55)
  - "Object Name Qualification in NonStop SQL/MX Release 2.1.1 and Later" (page 55)
  - "Sales Class" (page 107)
  - "Payroll Class" (page 109)

    - “Inventory Class” (page 110)
    - “LOWERPRICE Stored Procedure” (page 113)
    - “DAILYORDERS Stored Procedure” (page 115)
    - “MONTHLYORDERS Stored Procedure” (page 115)
    - “ADJUSTSALARY Stored Procedure” (page 117)
    - “EMPLOYEEJOB Stored Procedure” (page 118)
    - “SUPPLIERINFO Stored Procedure” (page 119)
    - “SUPPLYNUMBERS Stored Procedure” (page 120)
- Removed a note from the section “Portability” (page 20).
- Updated the Table 1 (page 32).
- Updated the section “Verifying the SQL/MX UDR Server” (page 33).
- Updated the section “Verifying the NonStop Server for Java” (page 34).
- Updated the section “Verifying the JDBC/MX Driver” (page 34).
- Updated the section “JREHOME Setting for the NonStop Server for Java” (page 35).
- Updated the section “Java Library Path for the JDBC/MX Native Library” (page 35).
- Added the section “Setting the JVM extensions location” (page 46).
- Removed a note from the section “Establishing Java Security” (page 46).
- Updated examples in the section “Null Input and Output” (page 52).
- Updated the section “Use of java.sql.Connection Objects” (page 53).
- Added a note to the section “Invoking SPJs in a NonStop ODBC/MX Client” (page 80).

Changes to the 691166-001 manual:

- Updated the section Verifying the SQL/MX UDR Server (page 33).
- Updated the section Naming the Procedure Label (page 66).
- Updated the section Calling an SPJ (page 71).
- Updated the section Transaction Statements or Methods in an SPJ Method (page 71).
- Added a new section Invoking SPJs in a Trigger (page 81).
- Updated the section CREATE PROCEDURE Statement (page 114).
- Updated the section CALL Statement to Invoke the SPJ (page 114).

Changes to the H06.16/J06.05 manual:

- Supported release statements have been updated to include J-series RVUs.
- Updated VALIDATEROUTINE2 internal SPJ details in Effect of Registering an SPJ (page 21).
- Updated Table 1: Software Products Required for SPJs (page 32).
- Updated the section Stored Procedure Result Sets (page 51).
- Updated the section Use of java.sql.Connection Objects (page 53).
- Added the section Specifying the Maximum Number of Result Sets (page 63).
- Added the section Using MXCI with SPJ RS (page 76).
- Added the section Output of the SHOWDDL Command of an SPJ with Result Sets (page 94).

- Added the section Using the EXPLAIN function with SPJ RS (page 103).
- Added a sample ORDERSUMMARY Stored Procedure (page 121).

Changes to the 540433-001 Manual

Section	New or Changed Information
Chapter 2: Getting Started	Updated Software Requirements (page 32) and Verifying Software Versions (page 33) and added Configuring the SPJ Environment on Systems Running H-Series RVUs (page 35)
Chapter 3: Writing SPJ Methods	Explained how to refer to the catalog and schema of database objects in Referring to Database Objects in an SPJ Method (page 55)
Chapter 6: Managing SPJs in NonStop SQL/MX	Removed information about migrating SPJs from SQL/MX Release 1.8 to SQL/MX Release 2.x because SQL/MX Release 1.8 is not supported on systems running H-series RVUs.

Document Organization

Chapter 1: Introduction	Introduces SPJs in NonStop SQL/MX, describes steps for using SPJs, describes the benefits of using SPJs, describes the process of registering and invoking SPJs in NonStop SQL/MX, and describes the SPJ environment.
Chapter 2: Getting Started	Describes the software requirements and system configuration for registering and invoking SPJs in NonStop SQL/MX.
Chapter 3: Writing SPJ Methods	Provides guidelines for writing and compiling a Java method to be used as the body of an SPJ.
Chapter 4: Registering SPJs in NonStop SQL/MX	Explains how to create an SPJ, drop an SPJ, and alter an SPJ in NonStop SQL/MX.
Chapter 5: Invoking SPJs in NonStop SQL/MX	Explains how to execute an SPJ by using the CALL statement in NonStop SQL/MX.
Chapter 6: Managing SPJs in NonStop SQL/MX	Describes management tasks related to SPJs, such as granting privileges for invoking SPJs, showing the SPJs that exist in a database, and using SPJs in a distributed database environment.
Chapter 7: Performance and Troubleshooting	Describes how to improve and monitor the performance of SPJs and provides guidelines for troubleshooting common problems.
Appendix A: Sample SPJs	Provides SPJ methods that demonstrate business logic in an SQL/MX database.

Examples in This Manual

The examples and sample SPJs in this manual refer to the SQL/MX sample database. To install the sample database, see the *SQL/MX Quick Start*. For information about the schema and tables of the sample database, see the *SQL/MX Reference Manual*.

Most of the examples and sample SPJs can query both SQL/MP and SQL/MX tables of the SQL/MX sample database. To query SQL/MP tables, use the NonStop SQL/MX Release 1.8 sample database. To query SQL/MX tables, use the NonStop SQL/MX Release 2.x sample database.

NOTE: The SQL/MX Release 2.x sample database uses SQL/MX format tables. To install the sample database, you must have a license to use SQL/MX DDL statements. To acquire this license, purchase product T0394. Without this product, you cannot install the sample database. An error message informs you that the system is not licensed.

For the entire set of sample SPJs, see Appendix A: Sample SPJs.

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

This requirement is described under Backup DAM Volumes and Physical Disk Drives

General Syntax Notation

The following list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS.

Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters.

Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type.

`Italic computer type` letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets.

Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$ terminal-name
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num ]
   [ -num]
   [ text]

K [ X | D ] address
```

{ } Braces.

A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line.

A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON] [SAVEABEND }
```

… Ellipsis.

An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…
[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation.

Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
LISTOPENS SU $process-name.# su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing.

Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing.

If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]…
```

!i and !o.

In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT ( segment-id !i , error ) ; !o
```

!i,o.

In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ; !i,o
```

!i:i.

In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ ( filename1:length !i:i , filename2:length ) ; !i:i
```

!o:i.

In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ ( filenum !i , [ filename:maxlen ] ) ; !o:i
```

Notation for Messages

The following list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text.

Bold text in an example indicates user input entered at the terminal. For example:

```
ENTER RUN CODE
?123
CODE RECEIVED: 123.00
```

The user must press the Return key after typing the input.

Nonitalic text.

Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters.

Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

```
p-register
process-name
```

[ ] Brackets.

Brackets enclose items that are sometimes, but not always, displayed. For example:

```
Event number=number [ Subject=first-subject-value ]
```

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list might be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL] [in SQL file system ]
```

{ } Braces.

A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list might be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object] [Operator] [Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown. }
```

| Vertical Line.

A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK] [Failed }
```

% Percent Sign.

A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
%B101111
%H2F
```

```
P=%p-register E=% e-register
```

Notation for Management Programming Interfaces

The following list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS.

Uppercase letters indicate names from definition files; enter these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters.

Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r.

The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME     token-type ZSPI-TYP-STRING. !r
```

!o.

The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER     token-type ZSPI-TYP-FNAME32. !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

Publishing History

This guide supports HP NonStop SQL/MX Release 3.2.1 until otherwise indicated by its replacement publication. The publication date and part number indicate the current edition of the document.

Part Number	Product Version	Publication Date
523727-003	NonStop SQL/MX Releases 2.0 and 2.1	June 2005
540433-001	NonStop SQL/MX Releases 2.0, 2.1, and 2.2	February 2006
540433-003	NonStop SQL/MX Releases 2.0, 2.1, 2.2, and 2.3	November 2008
691166-001	NonStop SQL/MX Release 3.2	August 2012
691166-002	NonStop SQL/MX Release 3.2.1	February 2013
691166-003	NonStop SQL/MX Release 3.2.1	June 2013

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

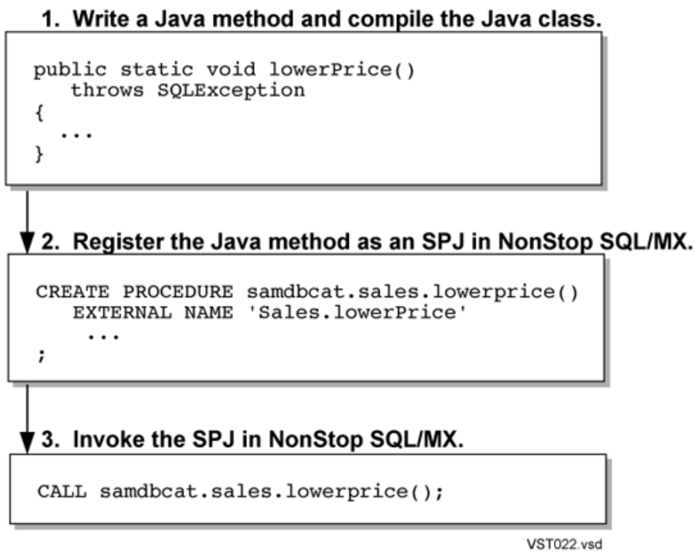
1 Introduction

This section introduces stored procedures in Java (SPJs) in NonStop SQL/MX and covers these topics:

What Is an SPJ?

A stored procedure is a type of user-defined routine (UDR) that operates within a database server. The database server contains the metadata of stored procedures and controls their execution. A stored procedure, which can perform SQL operations on a database, is invoked by a client application using an SQL CALL statement. Unlike a user-defined function, a stored procedure does not return a value directly to its caller. Instead, a stored procedure returns each output value to a host variable or dynamic parameter in its parameter list.

NonStop SQL/MX supports stored procedures written in the Java programming language. A stored procedure in Java (SPJ) is a Java method contained in a Java class, registered in SQL/MX system metadata tables, and invoked by NonStop SQL/MX when an application issues a CALL statement:



1. Write a Java method and compile the Java class.

```
public static void lowerPrice()
   throws SQLException
{
  ...
}
```

2. Register the Java method as an SPJ in NonStop SQL/MX.

```
CREATE PROCEDURE samdbcat.sales.lowerprice()
   EXTERNAL NAME 'Sales.lowerPrice'
    ...
;
```

3. Invoke the SPJ in NonStop SQL/MX.

```
CALL samdbcat.sales.lowerprice();
```

The body of a stored procedure consists of a `public`, `static` Java method that returns `void`. These methods, called *SPJ methods*, are contained in class files in the HP NonStop Open System Services (OSS) file system, and a group of class files or packages can be stored in a Java archive (JAR) file. The SQL statement, CREATE PROCEDURE, registers a Java method as a stored procedure in the database by storing its name, parameter types, location, and other metadata in SQL/MX system metadata tables. An SPJ method must be registered in NonStop SQL/MX before an SQL/MX application can call it. For more information about how SPJs operate in NonStop SQL/MX, see SPJs in NonStop SQL/MX (page 21).

NOTE: The SQL/MX implementation of SPJs complies mostly, unless otherwise specified, with SQL/JRT (Java Routines and Types), which extends the ANSI SQL/Foundation standard.

How Do I Use SPJs?

To create and invoke SPJs in NonStop SQL/MX:

1. Verify that you have the required software products installed on your HP NonStop system. See Verifying Software Versions (page 33).
2. Write and compile a static Java method to be used as an SPJ:

```
public class Payroll {

  public static void adjustSalary(BigDecimal empNum,
                                  double percent,
                                  BigDecimal[] newSalary)
    throws SQLException
  {
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement setSalary =
        conn.prepareStatement("UPDATE samdbcat.persnl.employee " +
                              "SET salary = salary * (1 + (? / 100)) " +
                              "WHERE empnum = ?");

    PreparedStatement getSalary =
        conn.prepareStatement("SELECT salary " +
                              "FROM samdbcat.persnl.employee " +
                              "WHERE empnum = ?");

    setSalary.setDouble(1, percent);
    setSalary.setBigDecimal(2, empNum);
    setSalary.executeUpdate();

    getSalary.setBigDecimal(1, empNum);
    ResultSet rs = getSalary.executeQuery();
    rs.next();
    newSalary[0] = rs.getBigDecimal(1);
    rs.close();

    conn.close();
  }
}
```

   Compile the Java source file that contains the method to produce a class file:

   `javac Payroll.java`

   For details, see Chapter 3: Writing SPJ Methods.

3. Configure the SPJ environment before registering or invoking SPJs:
   - If you installed the NonStop Server for Java in a nonstandard location, set the JREHOME location. See Setting the JREHOME Location (page 40).
   - If you installed the JDBC/MX driver in a nonstandard location, set the UDR extensions class path. See Setting the JDBC/MX Location (page 42).
   - If the SPJ methods refer to application classes outside their external paths, set those locations in the class path. See Setting the Class Path (page 43).
   - If necessary, set other JVM startup options for the SPJ environment. See Controlling JVM Startup Options (page 36).
   - If necessary, configure Java security for the SPJ environment. See Establishing Java Security (page 46).
4. As the super ID or schema owner, register each SPJ in NonStop SQL/MX by using the CREATE PROCEDURE statement:

```
CREATE PROCEDURE samdbcat.persnl.adju stsalary(IN empnum NUMERIC(4),
                                               IN percent FLOAT,
                                               OUT newsalary NUMERIC(8,2))
  EXTERNAL NAME 'Payroll.adjustSalary'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA;
```

For details, see Chapter 4: Registering SPJs in NonStop SQL/MX.

5. Grant privileges to users for invoking each SPJ:

```
GRANT EXECUTE
  ON PROCEDURE samdbcat.persnl.adjustsalary
  TO "HR.MGRNA", "HR.MGREU"
  WITH GRANT OPTION;

GRANT SELECT, UPDATE (salary)
  ON TABLE samdbcat.persnl.employee
  TO "HR.MGRNA", "HR.MGREU"
  WITH GRANT OPTION;
```

For details, see Granting Privileges for Invoking SPJs (page 83).

6. Invoke each SPJ by using the CALL statement:

```
CALL samdbcat.persnl.adjustsalary(29,2.5, ?);

NEWSALARY
------------

   139400.00

--- SQL operation complete.
```

For details, see Chapter 5: Invoking SPJs in NonStop SQL/MX.

7. Maintain privileges and manage the metadata of SPJs in the database environment. See Chapter 6: Managing SPJs in NonStop SQL/MX.
8. Monitor the performance of SPJs and resolve common problems with SPJs in NonStop SQL/MX. See Chapter 7: Performance and Troubleshooting.

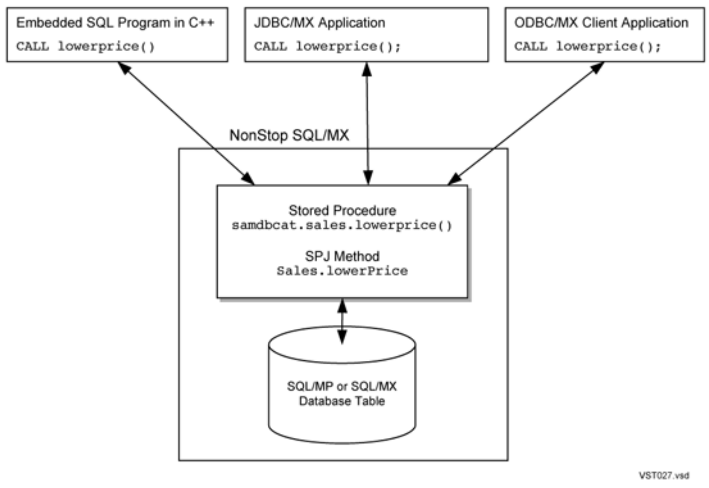
Benefits of SPJs

SPJs provide an efficient and secure way to implement business logic in an SQL/MX database. SPJs offer the advantages discussed next.

Java Methods Callable From NonStop SQL/MX

With support for SPJs, Java methods are callable from any SQL/MX application on a NonStop system. For example, you can invoke the same SPJ method from embedded SQL programs in C, C++, or COBOL and from HP NonStop ODBC/MX clients. By using NonStop SQL/MX to invoke Java methods, you can extend the functionality of an SQL/MX database and share business logic among different applications.

Figure 1 Different Applications Calling the Same SPJ



For more information, see Chapter 5: Invoking SPJs in NonStop SQL/MX.

Common Packaging Technique

Different applications can invoke the same SPJ to perform a common business function. By encapsulating business logic in an SPJ, you can maintain consistent database operations and avoid duplicating code in applications.

Applications that call SPJs are not required to know the structure of the tables that the SPJ methods access. If the table structure changes, you must change the SPJ methods but not necessarily the CALL statements within each application.

Security

By using SPJs, you can conceal sensitive business logic inside SPJ methods instead of exposing it in client applications. You can also grant privileges to execute an SPJ to specific users and restrict the privileges of other users. For more information, see Granting Privileges for Invoking SPJs (page 83).

Increased Productivity

Use SPJs to reduce the time and cost of developing and maintaining SQL/MX applications. By having several applications call the same SPJ, you need only change the SPJ method once when business rules or table structures change instead of changing every application that calls the SPJ.

Using the Java language to implement stored procedures increases development productivity. Given the popularity of the Java language, you can leverage the existing skill set of Java programmers to develop SPJs. The portability of the Java language enables you to write and compile Java class files for SPJs once and deploy them anywhere.

Portability

Because SPJ methods are written in Java, and SPJs conform to the ANSI SQL standard, SPJs are portable across different database servers. With minimal changes to SPJ methods, you can port existing Java classes from another database server to a NonStop server and register the methods as SPJs in NonStop SQL/MX. You can also port existing applications that call SPJs from other databases to NonStop SQL/MX with minimal changes to the CALL statements in the application.

SPJs in NonStop SQL/MX

This subsection explains the process and effect of registering and invoking SPJs in NonStop SQL/MX:

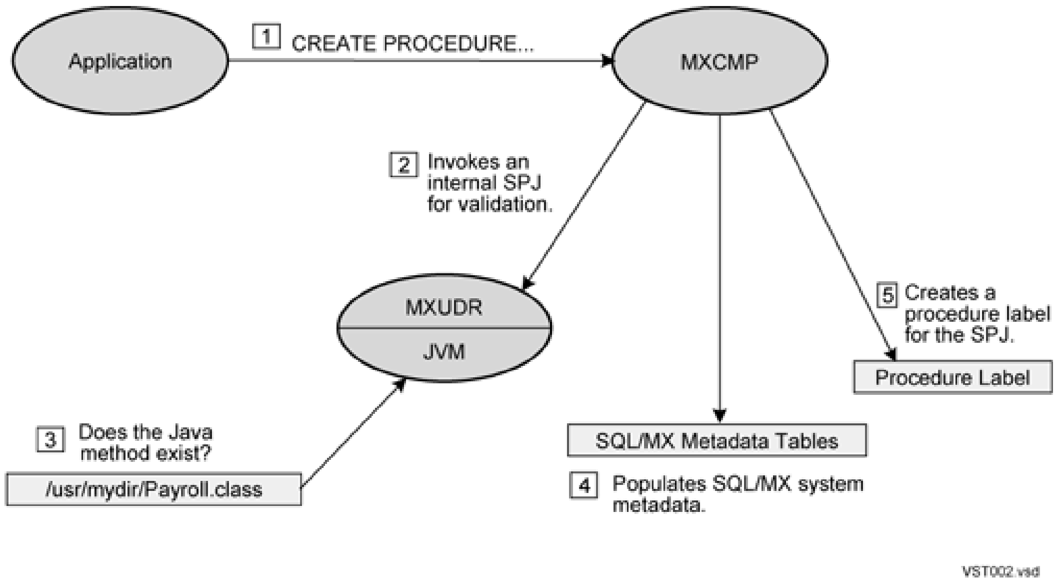
- Effect of Registering an SPJ (page 21)
- Effect of Invoking an SPJ (page 22)
- Invoking Different Types of SPJs (page 23)

Effect of Registering an SPJ

When you register an SPJ by using a CREATE PROCEDURE statement, NonStop SQL/MX verifies that the specified Java method exists and that its signature matches the SQL parameters of the stored procedure. After verifying the SPJ, NonStop SQL/MX stores information about the SPJ method, such as its name, location, and parameter types, in the system metadata tables. After you register an SPJ in NonStop SQL/MX, any SQL/MX application or interface with the appropriate permissions can call the SPJ to execute the SPJ method.

Figure 2 shows how NonStop SQL/MX processes a CREATE PROCEDURE statement.

Figure 2 Registering an SPJ


Application
1 CREATE PROCEDURE...
MXCMP
2 Invokes an internal SPJ for validation.
MXUDR
JVM
5 Creates a procedure label for the SPJ.
Procedure Label
3 Does the Java method exist?
/usr/mydir/Payroll.class
SQL/MX Metadata Tables
4 Populates SQL/MX system metadata.
VST002.vsd

1. The CREATE PROCEDURE statement that the application issues is processed by an SQL/MX compiler (MXCMP) process.
2. The catalog manager within the MXCMP process invokes aninternal SPJ named VALIDAT EROUTINE2, which is in SYSTEM_SQLJ_SCHEMA, and passes information from the CREATE PROCEDURE statement to it.
3. VALIDATEROUTINE2 validates the Java class and method specified in the CREATE PROCEDURE statement within an SQL/MX UDR server process (shown as the MXUDR executable in Figure 2).

   VALIDATEROUTINE2 uses the embedded Java virtual machine (JVM) within the SQL/MX UDR server process to verify that the Java method exists in the specified class and that its signature maps to the SQL parameters specified in the CREATE PROCEDURE statement. If the method exists and its signature maps to the SQL parameters, the SQL/MX UDR server returns a condition indicating success to the MXCMP process. Otherwise, the SQL/MX UDR server returns an error condition, and 4 do not complete.

NOTE: Systems running on J06.04 and earlier J-series RVUs or H06.15 and earlier H-series RVUs have VAILDATEROUTINE internal SPJ only, whereas systems running on J06.05 and later J-series RVUs or H06.16 and later H-series RVUs have both VALIDATEROUTINE and VALIDATEROUTINE2 internal SPJs.

Additionally, MXCMP released in H06.16 and later RVUs invokes internal SPJ VALIDATEROUTINE2 while MXCMP released in H06.15 or earlier RVUs invoke the VAILDATEROUTINE internal SPJ.

4. The catalog manager inserts information about the SPJ into the system metadata tables:
   - The ALL_UIDS table associates the SPJ with a unique identifier (UID).
   - The OBJECTS table contains information about each SPJ, such as its object name, unique identifier (UID), creation time, and ownership.
   - The ROUTINES table contains the attributes, such as external name and path, of each SPJ that is created in a catalog.
   - The COLS table contains the attributes of the individual parameters of an SPJ, one row per parameter.
   - The TEXT table contains the text of the Java encoded signature of an SPJ.
   - The TBL_PRIVILEGES table contains the ownership and granted privileges for each SPJ in a catalog.
   - The REPLICAS table contains information about the procedure label of an SPJ.

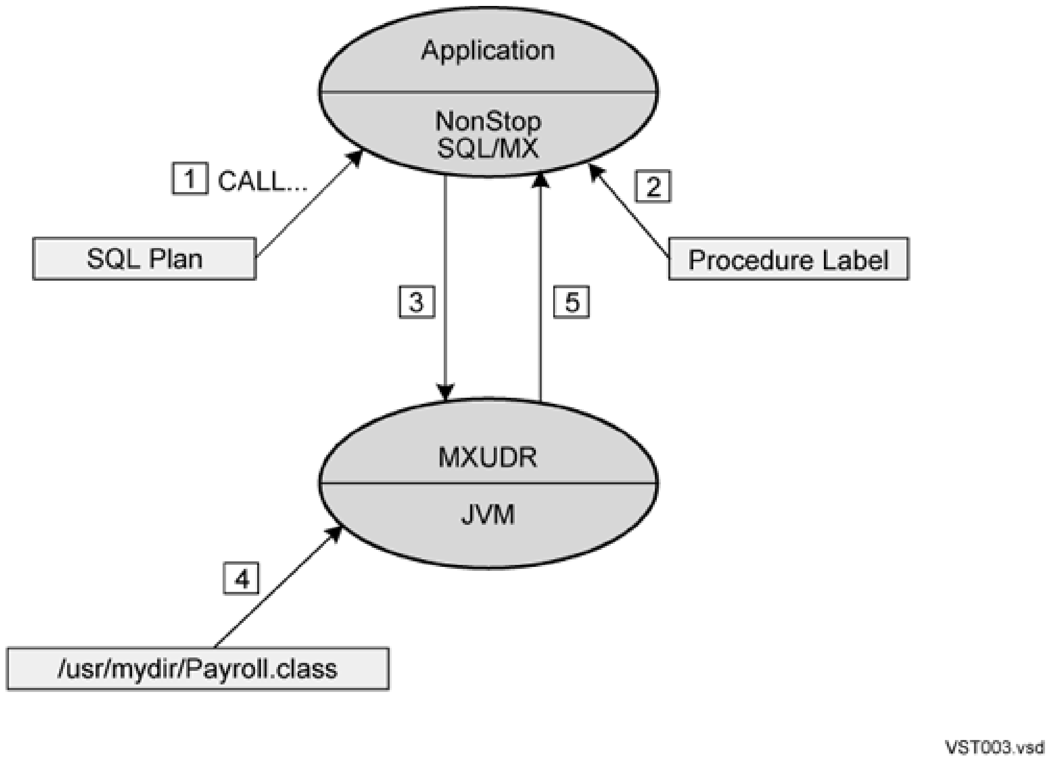
   For more information about these system metadata tables, see the *SQL/MX Reference Manual*.
5. The catalog manager also creates a procedure label owned by the user who issues the CREATE PROCEDURE statement.

   The procedure label is used internally by NonStop SQL/MX to track privileges on an SPJ. For more information, see Showing the Procedure Label (page 91).

Effect of Invoking an SPJ

When an application issues a CALL statement, the Java method of the invoked SPJ executes inside a JVM in an SQL/MX UDR server process. Figure 3 shows how SQL/MX processes a CALL statement.

Figure 3 Invoking an SPJ



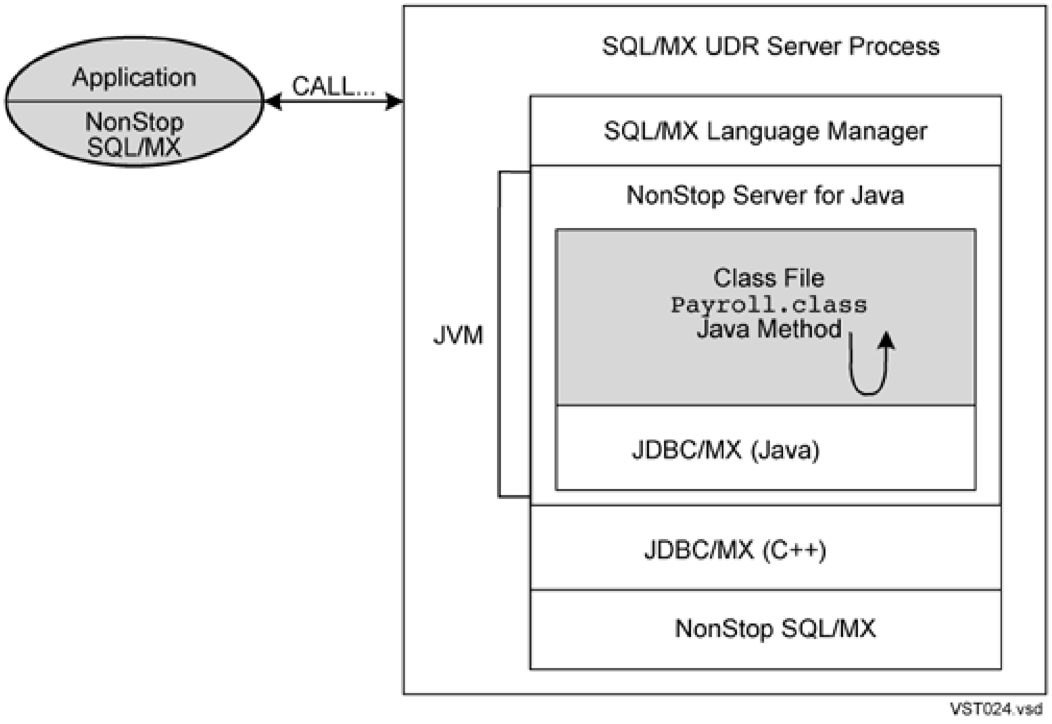
1. An SQL/MX application invokes an SPJ by issuing a CALL statement. The SQL/MX executor reads the compiled SQL plan for the CALL statement.
2. The SQL/MX executor checks the procedure label to verify that the caller has permission to invoke the SPJ.
3. If the caller has permission to invoke the SPJ, the SQL/MX executor starts sending messages to the SQL/MX UDR server (shown as the MXUDR executable in Figure 3) to execute the SPJ. The SQL/MX executor sends any input parameters that the application passes to the SPJ to the SQL/MX UDR server.
4. Inside the SQL/MX UDR server process, an SPJ class loader loads the class specified by the SPJ and executes the SPJ method.
5. The SQL/MX UDR server returns the output parameters of the SPJ to the SQL/MX executor, and the SQL/MX executor returns the output to the application. If exceptions occur during the execution of the Java method, the SQL/MX UDR server returns them to the SQL/MX executor as SQL diagnostics.

Invoking Different Types of SPJs

The next subsections show the layers of code within an SQL/MX UDR server process and how these layers interact when an application issues CALL statements that invoke different types of SPJ methods.

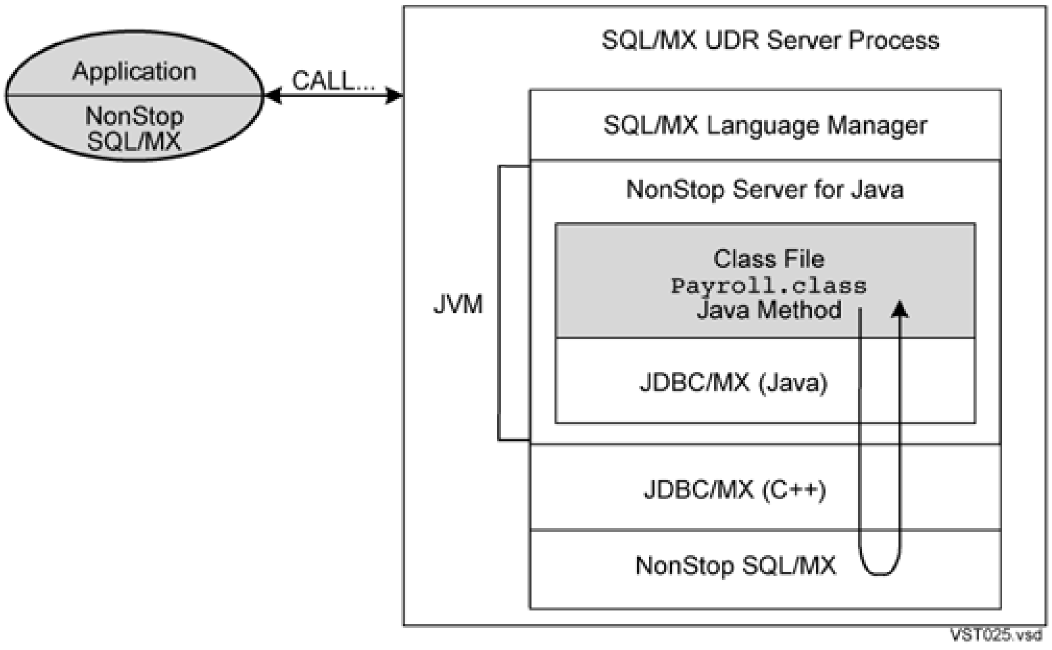
Pure Java Method

An SPJ can be based on a Java method from a pure Java program (that is, one that does not contain JDBC/MX method calls or SQLJ statements). A pure Java method does not directly perform database access operations. This type of SPJ method is loaded into the SPJ environment and executes its application logic by referring to its own class, other application classes, and Java system and extension classes:



JDBC/MX-Based Java Method

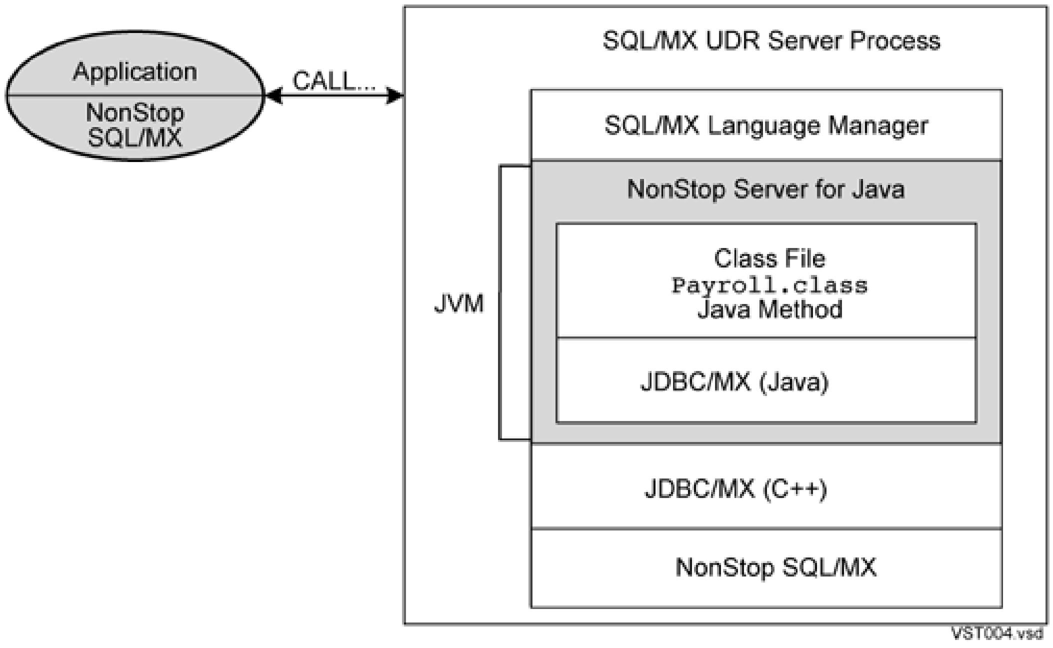
An SPJ method can be from a class file that uses standard JDBC method calls to access a NonStop SQL database. The pure Java and native layers of the JDBC/MX driver work together to process database requests issued by the SPJ method:



The SPJ Environment

NonStop SQL/MX processes CALL statements in an SPJ environment, which is hosted within an SQL/MX UDR server process. The SPJ environment includes an embedded JVM inside of which SPJ methods, invoked by CALL statements, execute, as shown in Figure 4.

Figure 4 SPJ Environment in NonStop SQL/MX



This subsection explains some of the ways in which the SPJ environment in NonStop SQL/MX differs from a typical Java environment:

- SQL/MX UDR Server Process (page 25)
- Multithreading in an SPJ Environment (page 27)
- Class Loaders in an SPJ Environment (page 28)
- Application Classes That an SPJ Can Access (page 29)
- Maintaining Class and JAR Files in an SPJ Environment (page 30)

SQL/MX UDR Server Process

The SQL/MX UDR server is a program executable named MXUDR in `$SYSTEM.SYSTEM`. When NonStop SQL/MX processes the first CALL statement of an application, it starts an SQL/MX UDR server process. An SQL/MX UDR server process hosts one SPJ environment and services one or more CALL statements of an SQL/MX application during its execution. When an application terminates, its SQL/MX UDR server processes also terminate.

For an SQL/MX application, the SQL/MX UDR server process that services the application can be single or multiple per connection to the database.

Single SQL/MX UDR Server Process Per Connection

An SQL/MX application that uses a single connection to the database has only one SQL/MX UDR server process service the application. All the CALL statements in such an application are processed in one SPJ environment.

Multiple SQL/MX UDR Server Processes Per Connection

An SQL/MX application that switches user identities or that uses different UDR_JAVA_OPTIONS settings has multiple SQL/MX UDR server processes service the application. The CALL statements in such an application are processed in multiple SPJ environments, with each CALL statement being processed in one SPJ environment.

Whenever there is a change in the user identity or in the UDR_JAVA_OPTIONS setting, a new SQL/MX UDR server process is launched. A different SPJ environment is created and the subsequent procedure calls are serviced using this new SQL/MX UDR process. See Figure 5 (page 27)

Applications that initiate multiple SQL/MX UDR server processes include NonStop ODBC/MX and JDBC/MX applications. NonStop ODBC/MX clients often switch user identities. Multithreaded JDBC/MX applications can use different UDR_JAVA_OPTIONS settings for different threads.

For these types of applications, new SQL/MX UDR server processes are started as needed, and stop when all the connections that were serviced by the SQL/MX UDR server process end. For more information, see Conditions Causing Initialization of an SPJ Environment (page 26).

The UDR_JAVA_OPTIONS default attribute controls the JVM startup options in an SPJ environment. For more information, see Controlling JVM Startup Options (page 36).

Conditions Causing Initialization of an SPJ Environment

When an application issues a CALL statement (that is, invokes an SPJ), the SQL/MX executor starts a new SQL/MX UDR server process and initializes a new SPJ environment if any of these conditions are met:

- An SPJ environment is not currently servicing the application (that is, the CALL statement is the first one executed by the application).
- The SPJ environments that are currently servicing the application are not associated with the current user identity.
- The CALL statement has a UDR_JAVA_OPTIONS setting (other than ANYTHING) that is different from the UDR_JAVA_OPTIONS setting in the SPJ environments that are servicing the application under the current user identity.

  See Determining the Uniqueness of UDR_JAVA_OPTIONS Settings (page 39).

Conditions Preventing Initialization of an SPJ Environment

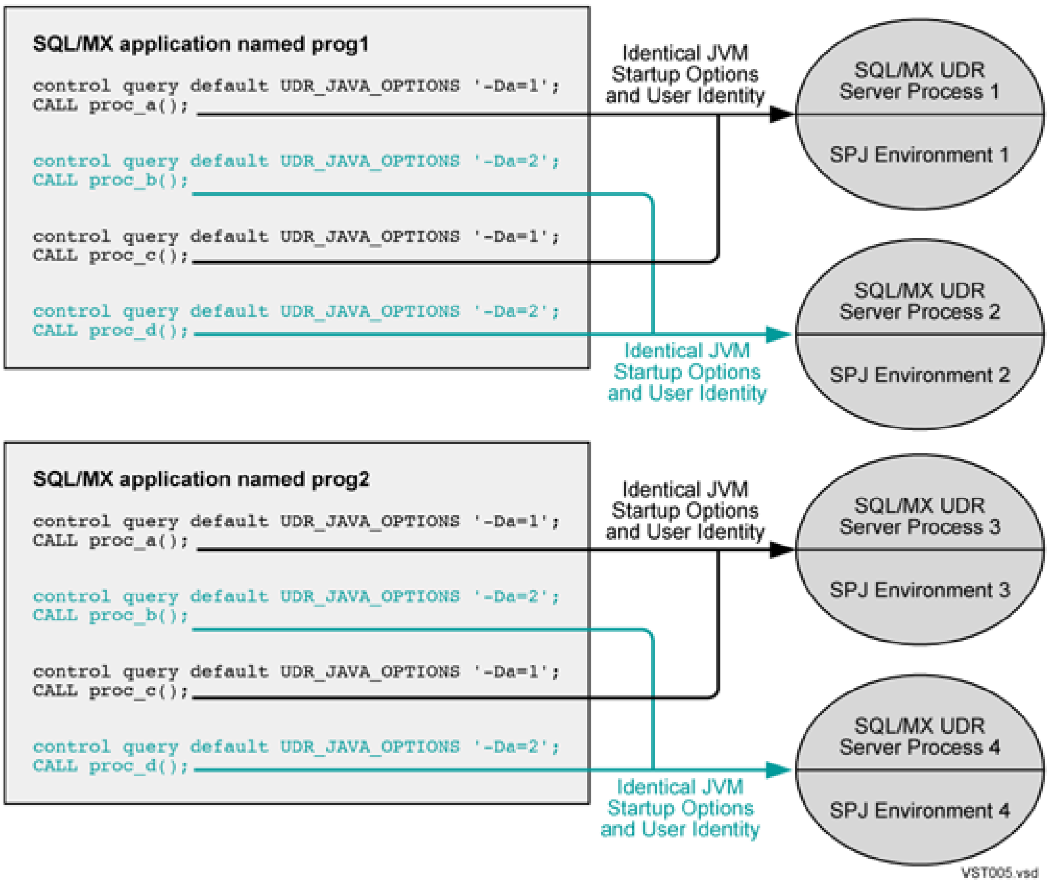
The execution of a CALL statement does not start a new SQL/MX UDR server process and initialize a new SPJ environment if one of these conditions is met:

- The CALL statement has a UDR_JAVA_OPTIONS setting (other than ANYTHING) that is the same as the UDR_JAVA_OPTIONS setting in one or more SPJ environments that are servicing the application under the current user identity.

  See Determining the Uniqueness of UDR_JAVA_OPTIONS Settings (page 39).
- The CALL statement has a UDR_JAVA_OPTIONS setting of ANYTHING, and one or more SPJ environments are currently servicing the application under the current user identity.

Figure 5 shows how SQL/MX UDR server processes and SPJ environments are initialized for an application.

Figure 5 Initialization of SQL/MX UDR Server Processes


SQL/MX application named prog1
control query default UDR_JAVA_OPTIONS '-Da=1';
CALL proc_a();
control query default UDR_JAVA_OPTIONS '-Da=2';
CALL proc_b();
control query default UDR_JAVA_OPTIONS '-Da=1';
CALL proc_c();
control query default UDR_JAVA_OPTIONS '-Da=2';
CALL proc_d();
Identical JVM Startup Options and User Identity
SQL/MX UDR Server Process 1
SPJ Environment 1
SQL/MX UDR Server Process 2
SPJ Environment 2
Identical JVM Startup Options and User Identity
SQL/MX application named prog2
control query default UDR_JAVA_OPTIONS '-Da=1';
CALL proc_a();
control query default UDR_JAVA_OPTIONS '-Da=2';
CALL proc_b();
control query default UDR_JAVA_OPTIONS '-Da=1';
CALL proc_c();
control query default UDR_JAVA_OPTIONS '-Da=2';
CALL proc_d();
Identical JVM Startup Options and User Identity
SQL/MX UDR Server Process 3
SPJ Environment 3
SQL/MX UDR Server Process 4
SPJ Environment 4
Identical JVM Startup Options and User Identity
VST005.vsd

Reinitialization of the SPJ Environment

In rare cases, the SQL/MX UDR server might crash unpredictably because of internal system errors or the application code. In such cases, the SPJ environment is reinitialized when a CALL statement is issued following the crash.

Multithreading in an SPJ Environment

Within an SPJ environment, NonStop SQL/MX manages a single thread of execution, even for multithreaded Java applications. The CALL statements in a multithreaded application can execute in a nonblocking manner, but the SPJ methods underlying those CALL statements execute serially within a given SPJ environment.

In SQL/MX Release 2.0 ABJ SPRs and later releases, the SQL/MX UDR server sets the default behavior to blocking mode. The blocking mode decreases CPU path length and improves performance in the SPJ environment. You can enable or disable nonblocking JDBC/MX in the SPJ environment by specifying JVM startup options. To specify JVM startup options in the SPJ environment, see Controlling JVM Startup Options (page 36).

For information about nonblocking JDBC/MX, see the *JDBC Type 2 Driver Programmer's Reference for SQL/MX*.

To support parallelism of CALL statements issued from a multithreaded application, use different UDR_JAVA_OPTIONS settings for each CALL statement. The execution of a CALL statement with a different UDR_JAVA_OPTIONS setting initiates a separate SQL/MX UDR server process with its own SPJ environment. These SPJ environments can process requests in parallel from a multithreaded application. For information about setting the UDR_JAVA_OPTIONS default attribute, see Controlling JVM Startup Options (page 36).

△ **CAUTION:** NonStop SQL/MX does not prevent SPJ methods from spawning other threads. Doing so is not advised and can lead to unpredictable results.

Class Loaders in an SPJ Environment

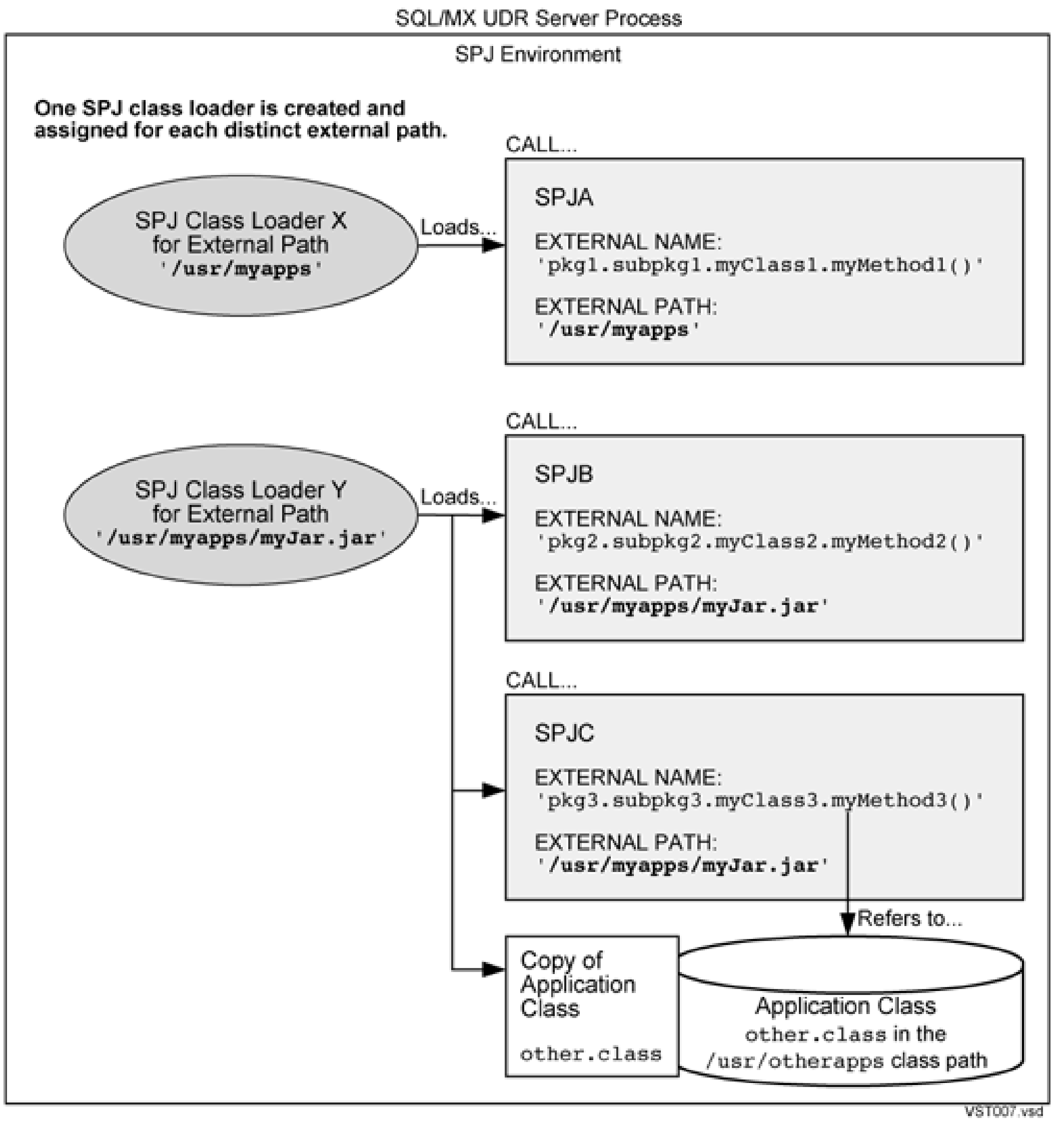
For system and extension classes, class loading in the SPJ environment is the same as other Java environments. By default, SPJ classes, like other Java programs, use the bootstrap class loader to load system classes, which are typically contained in the `rt.jar` file and are the Java core API. SPJ classes use the extension class loader to load classes from standard extension JAR files in the `jre/lib/ext` directory.

Class loading in the SPJ environment differs from other Java environments in that the SQL/MX UDR server manages certain aspects of class loading. The SQL/MX UDR server directs the system class loader, which typically manages an application's class path, to load classes in the SQL/MX language manager (`mxlangman.jar`), and in the JDBC/MX product file (`jdbcMx.jar`), and in the HP NonStop SQLJ product file (`sqlj.jar`), as needed.

The SQL/MX UDR server also creates and assigns one SPJ class loader for each distinct external path location, which you specify in the EXTERNAL PATH clause of a CREATE PROCEDURE statement when registering an SPJ. The SPJ class loader assigned to an external path becomes responsible for loading all SPJ classes in a directory or JAR file specified by that external path and for loading all application classes that are used by those SPJ classes. The SPJ class loader, instead of the system class loader, loads application classes from directories and JAR files specified in the class path.

In Figure 6, an SPJ method named `myMethod3()` in the `/usr/myapps/myJar.jar` external path accesses an application class named `other.class` in the `/usr/otherapps` class path. The SPJ class loader for the `/usr/myapps/myJar.jar` external path loads a copy of `other.class` in addition to the class file containing the SPJ method `myMethod3()`.

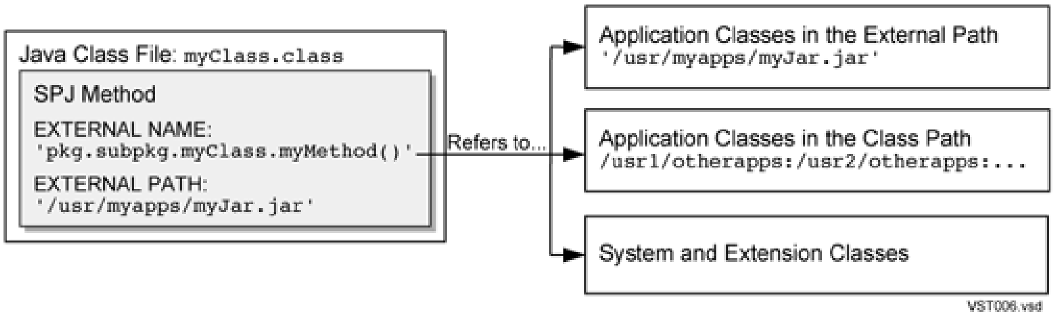
Figure 6 SPJ Class Loaders in an SQL/MX UDR Server Process


SQL/MX UDR Server Process
SPJ Environment
One SPJ class loader is created and assigned for each distinct external path.
SPJ Class Loader X for External Path '/usr/myapps'
Loads...
CALL...
SPJA
EXTERNAL NAME: 'pkg1.subpkg1.myClass1.myMethod1()'
EXTERNAL PATH: '/usr/myapps'
SPJ Class Loader Y for External Path '/usr/myapps/myJar.jar'
Loads...
CALL...
SPJB
EXTERNAL NAME: 'pkg2.subpkg2.myClass2.myMethod2()'
EXTERNAL PATH: '/usr/myapps/myJar.jar'
CALL...
SPJC
EXTERNAL NAME: 'pkg3.subpkg3.myClass3.myMethod3()'
EXTERNAL PATH: '/usr/myapps/myJar.jar'
Refers to...
Copy of Application Class other.class
Application Class other.class in the /usr/otherapps class path
VST007.vsd

Application Classes That an SPJ Can Access

A Java method that you register as an SPJ might need to access, either directly or indirectly, other Java classes to operate properly, as shown in Figure 7.

Figure 7 Java Classes That an SPJ Can Access


Java Class File: myClass.class
SPJ Method
EXTERNAL NAME: 'pkg.subpkg.myClass.myMethod()'
EXTERNAL PATH: '/usr/myapps/myJar.jar'
Refers to...
Application Classes in the External Path '/usr/myapps/myJar.jar'
Application Classes in the Class Path /usr1/otherapps:/usr2/otherapps:...
System and Extension Classes
VST006.vsd

To enable an SPJ method to refer to application classes, either put the application classes in the same external path as the SPJ class or specify the locations of the application classes in the class path.

Accessing Application Classes in the External Path

An SPJ method can access by default all application classes within its external path location, which you specify in the EXTERNAL PATH clause of a CREATE PROCEDURE statement when registering an SPJ. The external path is either an OSS directory or a JAR file path that contains the SPJ class file. See Specifying the External Path (page 66) .

If the external path is an OSS directory, only class files in the specified directory (or within packages in the specified directory) are considered to be in the external path. JAR files within the specified directory are not considered to be part of the external path.

If the external path is a JAR file path, only class files within the JAR file are considered to be in the external path. For example, if an SPJ class named `myClass` is contained in a JAR file named `myJar.jar`, the external path is the JAR file path, `/usr/myapps/myJar.jar`. All classes stored in the same JAR file as `myClass` are accessible by default to the SPJ method, `myMethod()`.

Accessing Application Classes Outside the External Path

For an SPJ method to access an application class outside its external path, you must set a search path in the SPJ environment called the class path. A class path can contain multiple directories and JAR file paths, which an SPJ class loader uses to search for referenced application classes.

During the execution of a CALL statement, if an SPJ method refers to an application class, an SPJ class loader first searches the external path for the class and, if the class is not found, it searches the class path.

For more information, see Setting the Class Path (page 43).

Maintaining Class and JAR Files in an SPJ Environment

You should not modify class and JAR files that are used by SPJs while active SPJ environments exist. If you update the SPJ classes or referenced application classes on disk, those changes might not be reflected in currently active SPJ environments, leading to unpredictable and undesirable behavior of CALL statements.

△ **CAUTION:** To prevent unpredictable and undesirable behavior of CALL statements, do not change SPJ classes or referenced application classes while SPJ environments are active. To update a class file or a JAR file that contains classes, stop the calling application, which ends the active SQL/MX UDR server process, install the updated class or JAR file, and then restart the application.

Consider the maintenance implications described next.

Updating the Java Class Files of an SPJ Method

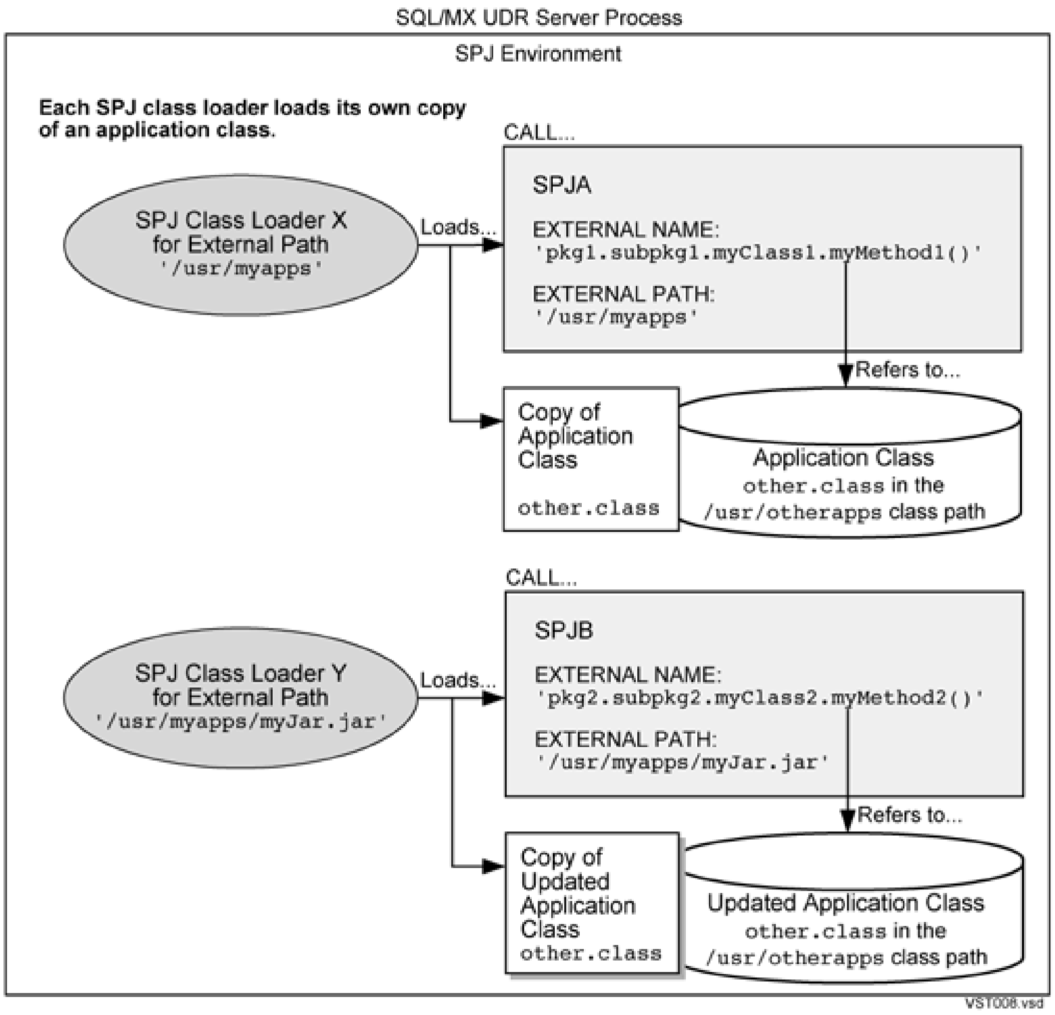
Suppose that you change an SPJ class while an SPJ environment remains active. After you change the Java class, subsequent calls to the SPJ in the same SPJ environment will cause NonStop SQL/MX to try to execute the old SPJ method, usually resulting in an error. Although you changed the Java class file on disk, the old class of the SPJ method remains cached in the JVM of the SPJ environment. After a class loader loads a class file, it does not reload the same class file unless garbage collection occurs.

Before installing an updated class file, or a JAR file that contains an updated class file, and before issuing a CREATE PROCEDURE statement, always stop the calling application, which ends the active SQL/MX UDR server process. For more information, see Altering an SPJ and Its Java Class (page 68).

Updating the Java Class Files of a Referenced Application Class

Suppose that two SPJs, each with a different external path location, depend on the same application class outside the external path but within the class path. Because the SPJ class loaders for each external path operate independently and do not share information, each loads its own copy of the application class at different times in the SPJ environment. Each time one of the SPJ class loaders loads an application class, the class loader creates and manages a new copy of the application class and its static variables. For an example, see Figure 8 (page 31).

Figure 8 Copies of Java Classes in SPJ Class Loaders



SQL/MX UDR Server Process
SPJ Environment
Each SPJ class loader loads its own copy of an application class.
CALL...
SPJA
EXTERNAL NAME: 'pkg1.subpkg1.myClass1.myMethod1()'
EXTERNAL PATH: '/usr/myapps'
SPJ Class Loader X for External Path '/usr/myapps'
Loads...
Refers to...
Copy of Application Class other.class
Application Class other.class in the /usr/otherapps class path
CALL...
SPJB
EXTERNAL NAME: 'pkg2.subpkg2.myClass2.myMethod2()'
EXTERNAL PATH: '/usr/myapps/myJar.jar'
SPJ Class Loader Y for External Path '/usr/myapps/myJar.jar'
Loads...
Refers to...
Copy of Updated Application Class other.class
Updated Application Class other.class in the /usr/otherapps class path
VST008.vsd

If an application class does not change on disk during an active SPJ environment, identical copies of the application class exist in memory and could waste resources. To conserve system resources, design your Java classes and directory structure so that fewer SPJ methods in different external paths rely on the same application class.

If an application class changes on disk during an active SPJ environment, copies of old and new versions of the application class could be in memory at the same time, leading to unpredictable and undesirable behavior of CALL statements. To prevent this problem, stop the calling application, which ends the SQL/MX UDR server process, install the updated class or JAR file, and then restart the application.

2 Getting Started

Before you can use SPJs in NonStop SQL/MX:

- Verify that your system meets software requirements.
- Install necessary software products.
- Configure the SPJ environment.

This section covers these topics:

Software Requirements

To use SPJs according to the instructions in this manual, you must have NonStop SQL/MX Release 3.x installed on a NonStop system. For detailed software requirements and installation instructions for NonStop SQL/MX, see the *SQL/MX Installation and Upgrade Guide*.

Table 1 lists the software products that you must install on a NonStop system to use SPJs in NonStop SQL/MX and whether you must relink the SQL/MX UDR server (the MXUDR executable) after installing those products.

Table 1 Software Products Required for SPJs

SQL/MX Product Version	Minimum Supported RVU	NonStop Server for Java*	JDBC/MX Driver*
NonStop SQL/MX Release 2.0	H06.03	T006H10 (SDK 1.4.2)	T1275H10 (equivalent to T1225V32)
NonStop SQL/MX Release 2.1	H06.04	T006H10 (SDK 1.4.2)	T1275H10 (equivalent to T1225V32)
NonStop SQL/MX Release 2.2	H06.04	T006H10 (SDK 1.4.2)	T1275H10 AAA (equivalent to T1225V32)
NonStop SQL/MX Release 2.3	H06.16	T2766 H50 (SDK 1.5.0)	T1275H50 AAJ
NonStop SQL/MX Release 3.0	H06.22	T2766 H50/T2766 H60	T1275H30
NonStop SQL/MX Release 3.1	H06.24	T2766 H50/T2766 H60	T1275H31
NonStop SQL/MX Release 3.2	H06.25	T2766 H50/T2766 H60	T1275H32
NonStop SQL/MX Release 3.2.1**	H06.26	T2766 H70 (NSJ7)***	T1275H32 AMU

*The product numbers are minimum versions on systems running H-series RVUs. See the README file on the NonStop Server for Java CD for installation instructions and the latest version requirements.

Table 1 Software Products Required for SPJs *(continued)*

SQL/MX Product Version	Minimum Supported RVU	NonStop Server for Java*	JDBC/MX Driver*

**Starting from NonStop SQL/MX Release 3.2.1, only NSJ7 or later versions of Java are supported.

***Distributed GC feature in NSJ7 is not supported for the SPJ.

NonStop Server for Java

To create and execute SPJs in NonStop SQL/MX, you must install the NonStop Server for Java on a NonStop system. Use the NonStop Server for Java to compile and execute Java applications. For details about this product, see the *NonStop Server for Java Programmer's Reference*. For installation instructions, see the README file on the product CD. If you install the NonStop Server for Java in a nonstandard location, see Setting the JREHOME Location (page 40). For version requirements, see Table 1.

JDBC/MX Driver

To create and execute SPJs in NonStop SQL/MX, you must install the JDBC/MX driver on a NonStop system. The JDBC/MX driver enables a Java application to use NonStop SQL/MX to access an SQL/MP or SQL/MX database. For details about this product, see the *JDBC Driver for SQL/MX Programmer's Reference*. For installation instructions, see the README file on the NonStop Server for Java product CD. If you install the JDBC/MX driver in a nonstandard location, see Setting the JDBC/MX Location (page 42). For version requirements, see Table 1.

Verifying Software Versions

Verify that you have compatible product versions of these necessary software products on a NonStop system:

- Verifying the SQL/MX UDR Server (page 33)
- Verifying the NonStop Server for Java (page 34)
- Verifying the JDBC/MX Driver (page 34)

Verifying the SQL/MX UDR Server

To display the product version of the SQL/MX UDR server, enter this `vproc` command at an OSS prompt:

```
vproc /G/system/system/mxudr
```

The command displays output similar to:

```
VPROC - T9617H01 - (01 FEB 2009) SYSTEM \YOSDEV1 Date 28 NOV 2012,
14:39:37
Copyright 2004 Hewlett-Packard Development Company, L.P.

/G/system/system/mxudr
     Binder timestamp: 26OCT2012 02:40:56
    Version procedure: T6520H02_01MAY2005_TFDS_SH_H02
    Version procedure: T1230H32_14FEB2013_APB_321_1025
    Version procedure: T8432H04_17FEB2012_CCPLMAIN
    Version procedure: T1231H32_14FEB2013_APB_321_1025
    Version procedure: T6520H02_01MAY2005_TFDSAPI_14JUN05_H02
    TNS/E Native Mode: runnable file
```

Compare the output of the command with the product version in the T1230 SOFTDOC file. If you have an incompatible product version of the SQL/MX UDR server on your system, see the *SQL/MX Installation and Upgrade Guide* for instructions about installing the correct release of NonStop SQL/MX. For version compatibility, See Table 1 (page 32).

On systems running H-series RVUs, notice that the NonStop Server for Java and JDBC/MX driver are not bound into the SQL/MX UDR server.

Verifying the NonStop Server for Java

To display the product version of the NonStop Server for Java in the Java environment of a NonStop system, enter this `vproc` command at an OSS prompt:

```
vproc /usr/tandem/nssjava/jdk170_h70/bin/java
```

If you have set the JREHOME environment variable for a nonstandard location of the NonStop Server for Java, enter:

```
vproc $JREHOME/../bin/java
```

The command displays output similar to:

```
VPROC - T9617H01 - (01 FEB 2009) SYSTEM \YOSDEV1 Date 28 NOV 2012,
14:47:37
Copyright 2004 Hewlett-Packard Development Company, L.P.

/usr/tandem/nssjava/jdk170_h70/bin/java
     Binder timestamp: 23OCT2012 20:04:54
    Version procedure: T2766H70_17FEB2013_jdk170_23Oct2012
    Version procedure: T8432H04_17FEB2012_CCPLMAIN
    TNS/E Native Mode: runnable file
```

Look for the product number of the JDK in the displayed output to determine the product version of the NonStop Server for Java that is installed on the system in the specified location.

Compare the output of the command with the product version in the README or SOFTDOC file of the NonStop Server for Java. If you have an incompatible product version of the NonStop Server for Java, obtain the correct version of the product and follow the installation instructions in the README file on the product CD. See also.

For version compatibility, See Table 1 (page 32)

Verifying the JDBC/MX Driver

To display the product version of the JDBC/MX driver in the Java environment of a NonStop system, enter this `java` command at an OSS prompt:

```
java -cp /usr/tandem/jdbcMx/current/lib/jdbcMx.jar JdbcMx
-version
```

The `-cp` option specifies the class path of the JDBC/MX driver, which, in this example, represents the standard location of the JDBC/MX driver. The location (standard or nonstandard) of the JDBC/MX driver must be specified in either the `-cp` option or the CLASSPATH environment variable.

If you have already set the class path of the JDBC/MX driver in the session, enter:

```
java JdbcMx -version
```

The command displays output similar to:

```
HP JDBC driver for NonStop(TM) SQL/MX Version T1275R32_20FEB2013_JDBCMXAMU_1029
```

The JDBCMX information indicates the product version of the JDBC/MX driver that is installed on the system in the specified location.

Compare the output of the command with the product version in the README or SOFTDOC file of the JDBC/MX driver. If you have an incompatible product version of JDBC/MX, obtain the correct version of the product and follow the installation instructions in the README file on the NonStop Server for Java product CD.

For version compatibility, See Table 1 (page 32)

Configuring the SPJ Environment on Systems Running H-Series RVUs

On systems running H-series RVUs, the libraries of the NonStop Server for Java and the JDBC/MX driver are not statically bound into the SQL/MX UDR server. Therefore, the SQL/MX UDR server depends on these settings to use the correct versions of the NonStop Server for Java and the JDBC/MX driver:

- JREHOME Setting for the NonStop Server for Java (page 35)
- Java Library Path for the JDBC/MX Native Library (page 35)
- UDR Extensions Class Path for the JDBC/MX JAR File (page 36)

JREHOME Setting for the NonStop Server for Java

To specify the location of the NonStop Server for Java for the SQL/MX UDR server, set the JREHOME location by using one of these methods, listed in order of precedence from highest to lowest:

1. In the UDR_JAVA_OPTIONS default attribute, set `-Dsqlmx.udr.jrehome` to the location of the NonStop Server for Java.
2. Set the JREHOME environment variable to the location of the NonStop Server for Java.
3. Let the SQL/MX UDR server use the location `/usr/tandem/java/jre`.

NOTE:

- Installation of NSJ7 does not automatically create the softlink `/usr/tandem/java`. If you have migrated all applications to NSJ7, then HP recommends you to create a softlink manually to point to the NSJ7 JDK/JRE installation (`/usr/tandem/nssjava/jdk170_h70`). Create the softlink `/usr/tandem/java` to point to the NSJ7 home directory as follows:

  ```
  $ ln -s /usr/tandem/nssjava/jdk170_h70 /usr/tandem/java
  ```

  After creating this softlink, applications that rely on the softlink might fail if they are not migrated to NSJ7.

- SPJs can be created through the NSM/web, and since the installation of NSJ7 does not automatically create the softlink `/usr/tandem/java`, you must use the method 1 or 3. Method 2 is not supported for NSM/web.

To use the JREHOME location, see Setting the JREHOME Location (page 40).

Java Library Path for the JDBC/MX Native Library

To load a native library, the JVM requires the directory of a native library to be specified in the `java.library.path` property. The same requirement applies to the JVM of the SQL/MX UDR server, which requires the directory of the JDBC/MX native library, `libjdbcMx.so`, to be in the `java.library.path`. The SQL/MX UDR server always appends the standard location of the JDBC/MX native library, `/usr/tandem/jdbcMx/current/lib`, to the end of the `java.library.path`.

To specify the location of the JDBC/MX native library, set the library path by using one of these methods, listed in order of precedence from highest to lowest:

1. In the UDR_JAVA_OPTIONS default attribute, set `-Djava.library.path` to the location of the JDBC/MX native library. The JVM of the SQL/MX UDR server uses this setting as the `java.library.path` property.
2. Set the _RLD_LIB_PATH environment variable to the location of the JDBC/MX native library. The JVM of the SQL/MX UDR server uses this setting as the `java.library.path` property.
3. Let the JVM of the SQL/MX UDR server use the standard location of the JDBC/MX native library, which the SQL/MX UDR server automatically appends to the `java.library.path` property.

To use UDR_JAVA_OPTIONS, see Controlling JVM Startup Options (page 36). To use the _RLD_LIB_PATH environment variable, see the *NonStop Server for Java 7.0 Programmer's Reference*.

UDR Extensions Class Path for the JDBC/MX JAR File

By default, NonStop SQL/MX loads the `jdbcMx.jar` file of JDBC/MX from the standard location of `/usr/tandem/jdbcMx/current/lib` into the SPJ environment. To use `jdbcMx.jar` in a nonstandard location, set the UDR extensions class path as you would on systems running G-series RVUs. For more information, see Setting the JDBC/MX Location (page 42).

Controlling JVM Startup Options

JVM startup options are passed to the Java application launcher (`java`) and influence the Java run-time environment. To specify JVM startup options for an SPJ environment, set the UDR_JAVA_OPTIONS default attribute. Use the UDR_JAVA_OPTIONS default attribute to:

- Override the NonStop Server for Java and JDBC/MX driver that are configured in the Java environment.
- Set class paths for an application that calls SPJs.
- Set Java system properties for an application that calls SPJs.
- Control the Java heap size of the SPJ environment.
- Troubleshoot the SPJ environment.

This subsection covers these topics:

- JVM Startup Options for All Processes on a Node (page 36)
- JVM Startup Options for Each SPJ Caller (page 37)
- Using Multiple UDR_JAVA_OPTIONS Settings in One Application (page 39)

For more information about the UDR_JAVA_OPTIONS default attribute, see the *SQL/MX Reference Manual*.

JVM Startup Options for All Processes on a Node

To set JVM startup options for all SPJ environments on a node, insert the UDR_JAVA_OPTIONS default attribute as a row in the SYSTEM_DEFAULTS table:

```
SET SCHEMA NONSTOP_SQLMX_node.SYSTEM_DEFAULTS_SCHEMA;

INSERT INTO SYSTEM_DEFAULTS
   (ATTRIBUTE, ATTR_VALUE)
   VALUES ('UDR_JAVA_OPTIONS',
           '-Djava.class.path=/usr/myclasses -Xmx32M');
```

The UDR_JAVA_OPTIONS setting in the SYSTEM_DEFAULTS table overrides the system-defined default setting, which is OFF. For more information, see the *SQL/MX Reference Manual*.

Scope of JVM Startup Options in the SYSTEM_DEFAULTS Table

A UDR_JAVA_OPTIONS setting in the SYSTEM_DEFAULTS table persists for all processes (or sessions) that run on the same node as the SYSTEM_DEFAULTS table. The insertion of the UDR_JAVA_OPTIONS setting into the SYSTEM_DEFAULTS table does not affect the current process. You must end and restart the process for the setting to take effect. In addition, the insertion does not affect previously compiled modules of an SQL/MX application unless you recompile those modules.

Precedence of JVM Startup Options

A UDR_JAVA_OPTIONS setting in a CONTROL QUERY DEFAULT statement takes precedence over the setting in the SYSTEM_DEFAULTS table. See JVM Startup Options for Each SPJ Caller (page 37).

Displaying the UDR_JAVA_OPTIONS in Effect for a Node

To show the UDR_JAVA_OPTIONS setting in effect for the system, enter the SHOWCONTROL ALL command in an MXCI session. For more information about SHOWCONTROL, see the *SQL/MX Reference Manual*.

JVM Startup Options for Each SPJ Caller

To set JVM startup options for each caller of an SPJ, use a CONTROL QUERY DEFAULT statement. A UDR_JAVA_OPTIONS setting in a CONTROL QUERY DEFAULT statement takes precedence over the system-defined default setting, which is OFF, and settings in the SYSTEM_DEFAULTS table.

For the effect of the UDR_JAVA_OPTIONS setting on different callers, see:

- Scope of JVM Startup Options in an SPJ Method (page 37)
- Scope of JVM Startup Options in Statically Compiled Applications (page 37)
- Scope of JVM Startup Options in Dynamically Compiled Applications (page 38)
- Scope of JVM Startup Options in an MXCI Session (page 39)

For information about the syntax of the CONTROL QUERY DEFAULT statement, see the *SQL/MX Reference Manual*.

Scope of JVM Startup Options in an SPJ Method

A UDR_JAVA_OPTIONS setting within an SPJ method does not affect the SPJ environment in which the SPJ is executing. It affects only the SPJ environment of CALL statements that the SPJ method executes. In general, avoid nesting CALL statements in an SPJ. For more information, see Nested Java Method Invocations (page 53).

Scope of JVM Startup Options in Statically Compiled Applications

The UDR_JAVA_OPTIONS setting in a statically compiled CONTROL QUERY DEFAULT statement affects statically compiled CALL statements in the line order scope of the application.

For example, this CONTROL QUERY DEFAULT statement in an embedded SQL program in C affects the maximum Java heap size of the SPJ environment in which the next statically compiled CALL statement executes:

```
/* Set application-specific system properties for
/* the SPJ environment. */
EXEC SQL CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS '-Xmx32M';

/* Call the stored procedure. */
EXEC SQL CALL samdbcat.sales.lowerprice();

/* Turn off the application-specific system properties for
/* the SPJ environment. */
EXEC SQL CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS 'OFF';

/* Call the stored procedure. */
EXEC SQL CALL samdbcat.sales.lowerprice();
...
```

CALL statements that are statically compiled after the CONTROL QUERY DEFAULT statement turns off the UDR_JAVA_OPTIONS setting execute in an SPJ environment without application-specific startup options.

For more information about the semantics of static CONTROL statements in embedded SQL programs, see the *SQL/MX Programming Manual for C and COBOL* or the *SQL/MX Programming Manual for Java*.

Scope of JVM Startup Options in Dynamically Compiled Applications

The UDR_JAVA_OPTIONS setting in a dynamically executed CONTROL QUERY DEFAULT statement affects CALL statements that are dynamically prepared after the CONTROL QUERY DEFAULT statement has executed.

For example, this CONTROL QUERY DEFAULT statement in an embedded SQL program in C sets application-specific system properties for the SPJ environment of the dynamically prepared CALL statement named `sqlstmt1`:

```
/* Build CALL statements in char buffers. */
strcpy(hv_sql_stmt1, "CALL samdbcat.sales.monthlyorders(?,?)");
                                                     /* IN, OUT */
strcpy(hv_sql_stmt2, "CALL samdbcat.persnl.adjustsalary(?,?,?)"
                                                     /* IN, IN, OUT */

for (i = 1; i <= 3; i++)
{
    /* Set application-specific system properties for
    /* the SPJ environment. */
    strcpy(hv_ctrl_stmt, "CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS '");
    printf("\nEnter JVM startup options: ");
    gets(attrval);
    strcat(hv_crtl_stmt, attrval);
    strcat(hv_ctrl_stmt, "'");

    /* Prepare and execute the CONTROL QUERY DEFAULT statement. */
    EXEC SQL EXECUTE IMMEDIATE :hv_ctrl_stmt;

    /* Prepare a CALL statement. */
    EXEC SQL PREPARE sqlstmt1 FROM :hv_sql_stmt1;

    printf("\nEnter a number for the month: ");
    scanf("%d", &hv_month_param1);


    /* Call the stored procedure. */
    EXEC SQL EXECUTE sqlstmt1
       USING :hv_month_param1      /* IN */
       INTO  :hv_ordernum_param2;  /* OUT */

    printf("\nThe number of orders during that month is %d.\n");

    /* Turn off the application-specific system properties for
    /* the SPJ environment. */
    strcpy(hv_ctrl_stmt, "CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS 'OFF'");
}

/* Prepare another CALL statement. */
EXEC SQL PREPARE sqlstmt2 FROM :hv_sql_stmt2;
...
/* Call the stored procedure. */
EXEC SQL EXECUTE sqlstmt2
   USING :hv_empnum_param1, :hv_percent_param2  /* IN, IN */
   INTO  :hv_newsalary_param3;                  /* OUT */
```

Dynamic CALL statements that are prepared outside of the loop, after the CONTROL QUERY DEFAULT statement turns off the UDR_JAVA_OPTIONS setting, execute in an SPJ environment without application-specific startup options.

For more information about the semantics of dynamic CONTROL statements in embedded SQL programs, see the *SQL/MX Programming Manual for C and COBOL* or the *SQL/MX Programming Manual for Java*.

Scope of JVM Startup Options in an MXCI Session

During an MXCI session, you can issue a CONTROL QUERY DEFAULT statement that sets the UDR_JAVA_OPTIONS default attribute. This setting applies only to CALL statements that you issue in MXCI after you issue the CONTROL QUERY DEFAULT statement.

For example, the first CONTROL QUERY DEFAULT statement sets the class path for the SPJ environment of subsequent CALL statements that you issue in MXCI. When you call the LOWERPRICE procedure after issuing the CONTROL QUERY DEFAULT statement, the JVM uses the `/usr/otherclasses` class path to search for and load classes that are not in the external path of that SPJ. The second CONTROL QUERY DEFAULT statement resets the UDR_JAVA_OPTIONS setting to the class path in effect, if any, at the start of the current MXCI session:

```
>>CALL samdbcat.sales.monthlyorders(1,?);

NUMBER
-----------

          1

--- SQL operation complete.

>>CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
                      '-Djava.class.path=/usr/myclasses';

--- SQL operation complete.

>>CALL samdbcat.sales.lowerprice();

--- SQL operation complete.

>>CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS RESET;

--- SQL operation complete.
```

For more information about MXCI, see the *SQL/MX Reference Manual*.

Displaying the UDR_JAVA_OPTIONS in Effect for an MXCI Session

To show the UDR_JAVA_OPTIONS setting in effect for a CONTROL QUERY DEFAULT statement issued in MXCI, enter the SHOWCONTROL command in the current MXCI session. For more information about SHOWCONTROL, see the *SQL/MX Reference Manual*.

Using Multiple UDR_JAVA_OPTIONS Settings in One Application

In the same application, you can set multiple UDR_JAVA_OPTIONS, each of which controls the JVM startup options in an SPJ environment. Two or more UDR_JAVA_OPTIONS settings are considered different or identical, depending on how you specify them in CONTROL QUERY DEFAULT statements. See Determining the Uniqueness of UDR_JAVA_OPTIONS Settings (page 39).

If you specify different UDR_JAVA_OPTIONS settings in the same application, be aware of the Performance Considerations of Using Multiple UDR_JAVA_OPTIONS Settings (page 40).

Determining the Uniqueness of UDR_JAVA_OPTIONS Settings

Two or more UDR_JAVA_OPTIONS settings are considered to be different when their character string literals do not match during a case-sensitive string comparison. The case, white space, and order in which the options appear within each character string literal must be the same for UDR_JAVA_OPTIONS settings to be considered identical.

For example, these UDR_JAVA_OPTIONS settings in separate CONTROL QUERY DEFAULT statements are considered to be the same:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Xmx32M -Djava.class.path=/usr/otherclasses';
```

```
CALL...
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Xmx32M -Djava.class.path=/usr/otherclasses';
CALL...
```

However, each of these UDR_JAVA_OPTIONS settings is different because of the different order of the options:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Xmx32M -Djava.class.path=/usr/otherclasses';
CALL...
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Djava.class.path=/usr/otherclasses -Xmx32M';
CALL...
```

These UDR_JAVA_OPTIONS settings are considered different because of differences in white space:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '  -Xmx32M  -Djava.class.path=/usr/otherclasses  ';
CALL...
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Xmx32M -Djava.class.path=/usr/otherclasses';
CALL...
```

Performance Considerations of Using Multiple UDR_JAVA_OPTIONS Settings

When an SQL/MX application executes CALL statements under different UDR_JAVA_OPTIONS settings, multiple SQL/MX UDR server processes service the application. NonStop SQL/MX maintains an SQL/MX UDR server process for each different set of UDR_JAVA_OPTIONS in an application. Each SQL/MX UDR server process contains its own embedded JVM.

CAUTION: The existence of multiple SQL/MX UDR server processes might significantly affect system performance. Therefore, use UDR_JAVA_OPTIONS settings in your application only when necessary.

For more information, see SQL/MX UDR Server Process (page 25).

Setting the JREHOME Location

By default, NonStop SQL/MX loads Java components from the standard location of the NonStop Server for Java into the SPJ environment. If you install the NonStop Server for Java in a nonstandard location, you must set the JREHOME location to the current installation directory of the NonStop Server for Java.

When an SQL/MX application starts with a JREHOME setting, NonStop SQL/MX loads the Java components from that JREHOME location into an SPJ environment created for the application.

To set the JREHOME location, use one of these approaches:

- Setting JREHOME by Using UDR_JAVA_OPTIONS (page 40)
- Setting the JREHOME Environment Variable (page 41)

Each approach has a different scope and influence on CALL or CREATE PROCEDURE statements.

Setting JREHOME by Using UDR_JAVA_OPTIONS

Use the UDR_JAVA_OPTIONS default attribute to set the JREHOME location in an SPJ environment for an application or for all processes running on the node. The UDR_JAVA_OPTIONS default attribute is particularly useful for NonStop ODBC/MX applications, which cannot use environment variables set in the OSS or Guardian environment.

To set the JREHOME location by using the UDR_JAVA_OPTIONS default attribute, use this attribute value:

```
'-Dsqlmx.udr.jrehome= java-installation-directory/jre'
```

The *java-installation-directory* is the path of the installation directory of the NonStop Server for Java.

△ **CAUTION:** NonStop SQL/MX maintains an SQL/MX UDR server process for each different set of UDR_JAVA_OPTIONS in an application. The existence of multiple SQL/MX UDR server processes might significantly affect system performance. Therefore, use UDR_JAVA_OPTIONS settings in your application only when necessary. For more information, see SQL/MX UDR Server Process (page 25).

Scope of JREHOME in UDR_JAVA_OPTIONS

A JREHOME location set by a CONTROL QUERY DEFAULT statement affects the SPJ environment of subsequent CALL or CREATE PROCEDURE statements. This JREHOME location setting persists until the SQL/MX UDR server process, which hosts the SPJ environment, ends. For more information, see SQL/MX UDR Server Process (page 25). For example, use a CONTROL QUERY DEFAULT statement to set the JREHOME location before a CALL statement in an application:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Dsqlmx.udr.jrehome=/usr/myjavadir/jre';
CALL...
```

A JREHOME setting inserted into the SYSTEM_DEFAULTS table affects all CALL or CREATE PROCEDURE statements that run on the system. This JREHOME setting persists until someone updates the UDR_JAVA_OPTIONS attribute in the SYSTEM_DEFAULTS table. For example, insert a row in the SYSTEM_DEFAULTS table to set the JREHOME location for all CREATE PROCEDURE statements that are registered on the system:

```
SET SCHEMA NONSTOP_SQLMX_node.SYSTEM_DEFAULTS_SCHEMA;

INSERT INTO SYSTEM_DEFAULTS
   (ATTRIBUTE, ATTR_VALUE)
   VALUES ('UDR_JAVA_OPTIONS',
           '-Dsqlmx.udr.jrehome=/usr/myjavadir/jre');
```

For more information about the scope of the UDR_JAVA_OPTIONS setting, see Controlling JVM Startup Options (page 36).

Precedence of JREHOME in UDR_JAVA_OPTIONS

The CONTROL QUERY DEFAULT setting overrides the UDR_JAVA_OPTIONS setting in the SYSTEM_DEFAULTS table. Both types of UDR_JAVA_OPTIONS settings take precedence over the JREHOME environment variable. See Setting the JREHOME Environment Variable (page 41).

Setting the JREHOME Environment Variable

Use the JREHOME environment variable to set the JREHOME location in an SPJ environment for a session in which applications that call or create SPJs run. Set the JREHOME environment variable in either the OSS or Guardian environment, depending on where you run applications that issue CALL or CREATE PROCEDURE statements.

NOTE: NonStop ODBC/MX applications cannot use environment variables set in the OSS or Guardian environment. See Setting JREHOME by Using UDR_JAVA_OPTIONS (page 40).

Scope of the JREHOME Environment Variable

The setting of the JREHOME environment variable persists for the current OSS or Guardian session and applies to all applications running in the current session.

Precedence of the JREHOME Environment Variable

The UDR_JAVA_OPTIONS setting takes precedence over the JREHOME environment variable. See Setting JREHOME by Using UDR_JAVA_OPTIONS (page 40).

Setting JREHOME in the OSS Environment

To set the JREHOME environment variable in the OSS environment, enter this command at an OSS prompt:

```
export JREHOME=java-installation-directory/jre
```

The *`java-installation-directory`* is the path of the installation directory of the NonStop Server for Java.

For example, this command sets the JREHOME location for SPJs invoked in the current OSS session:

```
export JREHOME=/usr/myjavadir/jre
```

For more information about the export command, see the *Open System Services Shell and Utilities Reference Manual*.

Setting JREHOME in the Guardian Environment

To set the JREHOME environment variable in the Guardian environment, enter this PARAM command at a Guardian prompt:

```
PARAM JREHOME java-installation-directory/jre
```

The *`java-installation-directory`* is the path of the installation directory of the NonStop Server for Java.

For example, this command sets the JREHOME location for SPJs invoked from embedded SQL programs in C, C++, or COBOL that will run in the current Guardian session:

```
PARAM JREHOME /usr/myjavadir/jre
```

For more information about the PARAM command, see the *TACL Reference Manual*.

Setting the JDBC/MX Location

By default, NonStop SQL/MX loads the `jdbcMx.jar` file of JDBC/MX from the standard location of `/usr/tandem/jdbcMx/current/lib` into the SPJ environment. In some cases, you might want to use another version of JDBC/MX that is compatible with NonStop SQL/MX but that is not installed in the standard location. To use `jdbcMx.jar` in a nonstandard location, you must set the UDR extensions class path to the JAR file path of `jdbcMx.jar`.

When an SQL/MX application starts with a UDR extensions class path setting for the JDBC/MX location, NonStop SQL/MX loads `jdbcMx.jar` from the specified location into an SPJ environment created for the application.

UDR Extensions Class Path

Before setting the JDBC/MX location, consider that:

- The UDR extensions include `jdbcMx.jar` and are not the same as the JVM extensions, which are typically located in `$JREHOME/lib/ext`.
- The SQL/MX UDR server has its own search path for UDR extensions, called the UDR extensions class path, and this class path contains a standard location for `jdbcMx.jar`.
- The UDR extensions class path is not the same as the class path. Do not use the class path to specify the location of a JDBC/MX driver in the SPJ environment. For more information about the class path, see Setting the Class Path (page 43).
- The `-Dsqlmx.udr.extensions` option adds a class path to the beginning of the search path of the SQL/MX UDR server. The `-Dsqlmx.udr.extensions` setting takes precedence over but does not override the search path.

NOTE: If you specify an alternate JDBC/MX location by using the `-Dsqlmx.udr.extensions` option and `jdbcMx.jar` does not exist at that location, NonStop SQL/MX uses the standard location for `jdbcMx.jar`.

Setting the JDBC/MX Location by Using UDR_JAVA_OPTIONS

Use the UDR_JAVA_OPTIONS default attribute to set the JDBC/MX location in an SPJ environment for an application or for all processes running on the node.

To set the JDBC/MX location by using the UDR_JAVA_OPTIONS default attribute, use this attribute value:

```
'-Dsqlmx.udr.extensions= jdbcmx-jar-filepath'
```

The *`jdbcmx-jar-filepath`* is the JAR file path of `jdbcMx.jar`.

CAUTION: NonStop SQL/MX maintains an SQL/MX UDR server process for each different set of UDR_JAVA_OPTIONS in an application. The existence of multiple SQL/MX UDR server processes might significantly affect system performance. Therefore, use UDR_JAVA_OPTIONS settings in your application only when necessary. For more information, see SQL/MX UDR Server Process (page 25).

Scope of the JDBC/MX Location in UDR_JAVA_OPTIONS

A JDBC/MX location set by a CONTROL QUERY DEFAULT statement affects the SPJ environment of subsequent CALL or CREATE PROCEDURE statements. This JDBC/MX location setting persists until the SQL/MX UDR server process, which hosts the SPJ environment, ends. For more information, see SQL/MX UDR Server Process (page 25). For example, use a CONTROL QUERY DEFAULT statement to set the JDBC/MX location before a CALL statement in an application:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Dsqlmx.udr.extensions=/usr/tandem/jdbcMx/T1225V311/lib/jdbcMx.jar';

CALL...
```

A JDBC/MX setting inserted into the SYSTEM_DEFAULTS table affects all CALL or CREATE PROCEDURE statements that run on the system. This JDBC/MX setting persists until someone updates the UDR_JAVA_OPTIONS attribute in the SYSTEM_DEFAULTS table. For example, insert a row in the SYSTEM_DEFAULTS table to set the JDBC/MX location for all CREATE PROCEDURE statements that are registered on the system:

```
SET SCHEMA NONSTOP_SQLMX_node.SYSTEM_DEFAULTS_SCHEMA;

INSERT INTO SYSTEM_DEFAULTS
   (ATTRIBUTE, ATTR_VALUE)
   VALUES ('UDR_JAVA_OPTIONS',
       '-Dsqlmx.udr.extensions=/usr/tandem/jdbcMx/T1225V311/lib/jdbcMx.jar');
```

For more information about the scope of the UDR_JAVA_OPTIONS setting, see Controlling JVM Startup Options (page 36).

Precedence of the JDBC/MX Location in UDR_JAVA_OPTIONS

The CONTROL QUERY DEFAULT setting overrides the UDR_JAVA_OPTIONS setting in the SYSTEM_DEFAULTS table.

Setting the Class Path

The class path contains an ordered list of directories and JAR files in which to search for Java class files. If an SPJ method refers to another application class (for example, loads a class, creates an object of a class, or invokes a method of another class), the other class must either be in the same external path location as the SPJ class or have its location specified in the class path.

When an SQL/MX application issues a CALL statement, a class loader in the SPJ environment uses the class path to locate and load Java classes outside the external path of the SPJ method. For more information, see Class Loaders in an SPJ Environment (page 28).

When an SQL/MX application issues a CREATE PROCEDURE statement, the JVM resolves references to other Java classes in the signature of the SPJ method but not inside the body of the SPJ method. For example, the JVM might encounter a user-defined exception class in the `throws` clause of the signature, as shown:

```
public static void lowerPrice()
   throws pkg.subpkg.myException
```

If the user-defined exception class exists outside the external path, the location of the exception class must be listed in the class path.

To set the class path in an SPJ environment, use these approaches:

- Setting the Class Path by Using UDR_JAVA_OPTIONS (page 44)
- Setting the CLASSPATH Environment Variable (page 45)

Each approach has a different scope and influence on CALL or CREATE PROCEDURE statements.

NOTE: To specify a nonstandard location of the JDBC/MX driver in the SPJ environment, use the UDR extensions class path instead of the class path. Set the UDR extensions class path by using the `-Dsqlmx.udr.extensions` option in the UDR_JAVA_OPTIONS default attribute. See Setting the JDBC/MX Location (page 42).

Setting the Class Path by Using UDR_JAVA_OPTIONS

Use the UDR_JAVA_OPTIONS default attribute to set the class path in an SPJ environment for an application or for all processes running on the node. The UDR_JAVA_OPTIONS default attribute is particularly useful forNonStop ODBC/MX applications, which cannot use environment variables set in the OSS or Guardian environment.

To set the class path by using the UDR_JAVA_OPTIONS default attribute, use this attribute value:

```
'-Djava.class.path=path1[{:path2}...]'
```

`path1` and *`path2`* are paths to JAR files or top-level directories where Java classes or package directories exist. The paths must not include the package name.

CAUTION: NonStop SQL/MX maintains an SQL/MX UDR server process for each different set of UDR_JAVA_OPTIONS in an application. The existence of multiple SQL/MX UDR server processes might significantly affect system performance. Therefore, use UDR_JAVA_OPTIONS settings in your application only when necessary. For more information, see SQL/MX UDR Server Process (page 25).

Scope of the Class Path in UDR_JAVA_OPTIONS

A class path set by a CONTROL QUERY DEFAULT statement affects the SPJ environment of subsequent CALL or CREATE PROCEDURE statements. This class path setting persists until the SQL/MX UDR server process, which hosts the SPJ environment, ends. For more information, see SQL/MX UDR Server Process (page 25). For example, use a CONTROL QUERY DEFAULT statement to set the class path before a CALL statement in an application:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
   '-Djava.class.path=/usr/otherclasses:/usr/otherapps/myJar.jar';
CALL...
```

A class path inserted into the SYSTEM_DEFAULTS table affects all CALL or CREATE PROCEDURE statements that run on the system. This class path setting persists until someone updates the UDR_JAVA_OPTIONS attribute in the SYSTEM_DEFAULTS table. For example, insert a row in the

SYSTEM_DEFAULTS table to set the class path for all CREATE PROCEDURE statements that are registered on the system:

```
SET SCHEMA NONSTOP_SQLMX_node.SYSTEM_DEFAULTS_SCHEMA;

INSERT INTO SYSTEM_DEFAULTS
   (ATTRIBUTE, ATTR_VALUE)
   VALUES ('UDR_JAVA_OPTIONS',
           '-Djava.class.path=/usr/otherclasses:/usr/otherapps/myJar.jar');
```

For more information about the scope of the UDR_JAVA_OPTIONS setting, see Controlling JVM Startup Options (page 36).

Precedence of the Class Path in UDR_JAVA_OPTIONS

The CONTROL QUERY DEFAULT setting overrides the UDR_JAVA_OPTIONS setting in the SYSTEM_DEFAULTS table. Both types of UDR_JAVA_OPTIONS settings take precedence over the CLASSPATH environment variable. See Setting the CLASSPATH Environment Variable (page 45).

Setting the CLASSPATH Environment Variable

Use the CLASSPATH environment variable to set the class path in an SPJ environment for a session in which applications that call or create SPJs run. Set the CLASSPATH environment variable in either the OSS or Guardian environment, depending on where you run applications that issue CALL or CREATE PROCEDURE statements.

NOTE: NonStop ODBC/MX applications cannot use environment variables set in the OSS or Guardian environment. See Setting the Class Path by Using UDR_JAVA_OPTIONS (page 44).

Scope of the CLASSPATH Environment Variable

The setting of the CLASSPATH environment variable persists for the current OSS or Guardian session and applies to all applications running in the current session.

Precedence of the CLASSPATH Environment Variable

The UDR_JAVA_OPTIONS setting takes precedence over the CLASSPATH environment variable. See Setting the Class Path by Using UDR_JAVA_OPTIONS (page 44).

Setting CLASSPATH in the OSS Environment

To set the CLASSPATH environment variable in the OSS environment, enter this command at an OSS prompt:

```
export CLASSPATH=["][$CLASSPATH:]path1[{:path2}...]["]
```

path1 and *path2* are paths to JAR files or top-level directories where Java classes or package directories exist. The paths must not include the package name.

For example, this command sets the class path for SPJs invoked in the current OSS session:

```
export CLASSPATH="$CLASSPATH:/usr/otherapps/myJar.jar:
                             /usr/otherclasses"
```

For more information about the `export` command, see the *Open System Services Shell and Utilities Reference Manual*.

Setting CLASSPATH in the Guardian Environment

To set the CLASSPATH environment variable in the Guardian environment, enter this PARAM command at a Guardian prompt:

```
PARAM CLASSPATH path1[{: path2}...]
```

`path1` and `path2` are paths to JAR files or top-level directories where Java classes or package directories exist. The paths must not include the package name.

For example, this command sets the class path for SPJs invoked from embedded SQL programs in C, C++, or COBOL that will run in the current Guardian session:

```
PARAM CLASSPATH /usr/otherapps/myJar.jar:/usr/otherclasses
```

For more information about the PARAM command, see the *TACL Reference Manual*.

Setting the JVM extensions location

JRE supports the use of optional packages through its "extension mechanism". An optional package, also known as JVM extensions, is a group of packages combined in one or more JAR files. These JAR files implement an API that extends the Java platform. The virtual machine can find and load optional package classes without them being on the class path.

Java platform extensions are typically located in `$JREHOME/lib/ext`. If the optional packages are not present in this standard location, then include the location of the optional packages that are referred by the core Java API using the UDR extensions class path. SQL/MX UDR server uses the UDR extensions class path to search for JVM extensions.

NOTE: The class path `'java.class.path'` cannot be used for setting the location of optional packages.

Setting the JVM extensions location by Using UDR_JAVA_OPITIONS

Use the UDR_JAVA_OPTIONS default attribute to set the UDR extensions location in an SPJ environment for an application or for all processes running on the node.

```
'-Dsqlmx.udr.extensions=jvm-extensions-path'
```

The `jvm-extensions-path` is the JAR file path of the optional extension package.

Installing JAR Files in NonStop SQL/MX

NonStop SQL/MX supports SPJ methods that are packaged in JAR files but does not support the INSTALL_JAR, REMOVE_JAR, REPLACE_JAR, and ALTER_JAR_PATH procedures from SQL/JRT of the ANSI SQL/Foundation standard.

To use a Java method in a JAR file as the body of an SPJ:

1. Place the JAR file in an OSS directory where you want to register the SPJ.

   For example, this OSS command copies the `myJar.jar` file from a private location of `/usr/myfiles` to the public location of `/usr/spjfiles`:

   ```
   cp /E/DEV/usr/mydir/myJar.jar \
      /E/PROD/usr/spjfiles/myJar.jar
   ```

2. Issue a CREATE PROCEDURE statement to register the SPJ, using the full OSS path of the JAR file in the EXTERNAL PATH clause:

   ```
   CREATE PROCEDURE samdbcat.sales.lowerprice()
      EXTERNAL NAME 'pkg.subpkg.Sales.lowerPrice'
      EXTERNAL PATH '/usr/spjfiles/myJar.jar'
      ...
   ```

For more information, see Creating an SPJ (page 59).

Establishing Java Security

Java has built-in security features, such as the Java security manager, that protect against unauthorized use of or access to the system. With Java security enabled, the Java security manager protects the SPJ environment by restricting access to system resources.

By using a policy file, you can configure the Java security manager so that specific Java classes and methods can perform restricted operations, such as accessing a directory or network address.

Using UDR_JAVA_OPTIONS to Enable Java Security

By default, Java security is disabled in the SPJ environment of an SQL/MX UDR server process. To enable Java security in the SPJ environment, use this UDR_JAVA_OPTIONS attribute value:

```
'-Djava.security.manager-Djava.security.policy=
   /usr/tandem/sqlmx/udr/mxlangman.policy'
```

For example, set the UDR_JAVA_OPTIONS default attribute in a CONTROL QUERY DEFAULT statement as:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
  '-Djava.security.manager -Djava.security.policy=
   /usr/tandem/sqlmx/udr/mxlangman.policy';
```

For other ways of setting the UDR_JAVA_OPTIONS default attribute, see Controlling JVM Startup Options (page 36).

The UDR_JAVA_OPTIONS setting enables a Java security manager in the SPJ environment. The Java security manager first loads the default, system-wide Java policy file, `java-installation-directory/jre/lib/security/java.policy`, and then loads the SPJ policy file specified by `-Djava.security.policy`. Java security remains enabled for the duration of the SPJ environment until the SQL/MX UDR server process, which hosts the SPJ environment, ends.

You should specify only one SPJ policy file in a UDR_JAVA_OPTIONS setting. Otherwise, the last policy file listed takes effect. In this example, the `mypolicy.policy` file takes precedence over the `mxlangman.policy` file:

```
CONTROL QUERY DEFAULT UDR_JAVA_OPTIONS
  '-Djava.security.manager -Djava.security.policy=
   /usr/tandem/sqlmx/udr/mxlangman.policy
   -Djava.security.policy=/usr/myfiles/mypolicy.policy';
```

SPJ Policy File and Required Permissions

The default SPJ policy file, `mxlangman.policy` in the `/usr/tandem/sqlmx/udr` directory, contains these permissions:

```
grant codeBase "file:/usr/tandem/sqlmx/udr/mxlangman.jar" {
   permission java.security.AllPermission;
};
grant codeBase
 "file:/usr/tandem/jdbcMx/current/lib/jdbcMx.jar" {
   permission java.security.AllPermission;
};
```

You can use the default SPJ policy file, `mxlangman.policy`, as is, reconfigure it, or use your own policy file. The policy file that you specify must contain certain permissions for SPJs to operate properly in the SPJ environment. Details are discussed next.

Permissions for the SQL/MX Language Manager

The SQL/MX language manager is a key component because it loads, invokes, and unloads SPJs in an SQL/MX UDR server process. The `mxlangman.jar` file contains Java bytecode that implements part of the SQL/MX language manager.

You must grant these permissions in the SPJ policy file for the SQL/MX language manager to operate properly:

```
grant codeBase "file:/usr/tandem/sqlmx/udr/mxlangman.jar" {
   permission java.security.AllPermission;
};
```

If the SPJ policy file does not contain these permissions, all CALL statements fail and return errors that describe a security-related problem with the SQL/MX language manager.

Permissions for the JDBC/MX Driver

If you plan to call SPJs that access a database (that is, JDBC/MX-based SPJs), you must grant these permissions in the SPJ policy file for JDBC/MX product to operate properly:

```
grant codeBase
 "file:/usr/tandem/jdbcMx/current/lib/jdbcMx.jar" {
   permission java.security.AllPermission;
};
```

Specify the standard location (`/usr/tandem/jdbcMx/current/lib/ jdbcMx.jar`) or a nonstandard location that you specify in the UDR extensions class path. For more information, see Setting the JDBC/MX Location (page 42).

For more information about JDBC/MX, see the *JDBC Driver for SQL/MX Programmer's Reference*.

Permissions for the Java System and Extension Classes

Java system classes, such as Java core API classes, and standard extension packages are granted permissions by the default, system-wide Java policy file, `java.policy`, not the SPJ policy file. The Java core API classes in the NonStop Server for Java are always granted all permissions. Extension packages in *`java-installation-directory`*`/jre/lib/ext` are typically granted all permissions.

For information about `java.policy`, see the Java documentation.

Permissions for the SPJ Method

To allow an SPJ method, or any application class on which the SPJ method depends, to perform a restricted operation when Java security is enabled in the SPJ environment, you must grant the appropriate permissions to the codebase of that Java method. The codebase identifies the location of the class or JAR file that contains the Java method. If you do not grant the appropriate permissions in the SPJ policy file, the invoked SPJ does not behave as desired, or the CALL statement fails to execute.

For example, suppose that you want to invoke an SPJ that consists of an SPJ method that reads OSS files. If you rely on the default permissions in the SPJ policy file, `mxlangman.policy`, the CALL statement that invokes the SPJ fails and returns an error because reading a file is a restricted operation when Java security is enabled. To allow the SPJ method to read OSS files while Java security is enabled, add this grant statement to the SPJ policy file:

```
grant codeBase "file:/usr/mydir/myJar.jar" {
   permission java.io.FilePermission "/usr/ossfiles",
              "read,write";
};
```

In the example, the SPJ method that reads the OSS files is packaged in a JAR file, `myJar.jar`, within the `/usr/mydir` directory. To grant this permission to all the classes in the `/usr/mydir` directory, including the JAR file, add this grant statement to the SPJ policy file:

```
grant codeBase "file:/usr/mydir/*" {
   permission java.io.FilePermission "/usr/ossfiles",
              "read,write";
};
```

For information about policy file syntax and Java security, see the Java documentation.

3 Writing SPJ Methods

Before you can create an SPJ in NonStop SQL/MX, you must write and compile the Java method to be used as the body of the SPJ. The Java methods that you use for the body of an SPJ are called SPJ methods.

This section requires a familiarity with writing and compiling Java programs and covers these topics:

Guidelines for Writing SPJ Methods

Follow these guidelines when you write SPJ methods to be used as SPJs in NonStop SQL/MX:

Signature of the Java Method

A Java method that you use as an SPJ must have this general signature:

```
public static void myMethodName ( java-parameter-list)
```

Public Access and Static Modifiers

The Java method must be defined as `public` and `static`. If a method is `private` or `protected`, NonStop SQL/MX is unable to find the Java method specified by the CREATE PROCEDURE statement and returns an error. The Java method must be defined as `static` so that the method can be invoked without having to instantiate its class.

Void Return Type

The return type of the Java method must be `void`. The method must not return a value directly to the caller.

Java Parameters

The parameter types in the Java signature must correspond to the SQL parameters of the stored procedure that you are planning to create. For type mappings, Table 2.

Table 2 Mapping of Java Data Types to SQL/MX Data Types

Java D ata Type	Maps to SQL/MX Data Type...
`java.lang.String`	`CHAR[ACTER] CHAR[ACTER] VARYINGVARCHARPIC[TURE] XNCHAR NCHAR`

Table 2 Mapping of Java Data Types to SQL/MX Data Types *(continued)*

Java D ata Type	Maps to SQL/MX Data Type...
	VARYING NATIONAL CHAR[ACTER] NATIONAL CHAR[ACTER] VARYING
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.math.BigDecimal	NUMERICDEC[IMAL]PIC[TURE] S9
short	SMALLINT
int or java.lang.Integer*	INT[EGER]
long or java.lang.Long*	LARGEINT
double or java.lang.Double*	FLOAT
float or java.lang.Float*	REAL
double or java.lang.Double*	DOUBLE PRECISION

* Choose a Java wrapper class if you plan to pass null values as arguments to or from the method. See Null Input and Output (page 52).

Output parameters in the Java signature must be arrays (for example, `int[]` or `String[]`) that accept only one value in the first element of the array at index 0. For more information, see Returning Output Values From the Java Method.

Returning Output Values From the Java Method

Output parameters in the Java signature must be parameter arrays that accept one value in the first element of the array at index 0. This subsection covers these topics related to output parameters:

- Using Arrays for Output Parameters (page 50)
- Type Mapping of Output Parameters (page 51)
- Stored Procedure Result Sets (page 51)

Using Arrays for Output Parameters

You must use arrays for the output parameters of a Java method because of how Java handles the arguments of a method. Java supports arguments that are passed by value to a method and does not support arguments that are passed by reference. As a result, Java primitive types can be passed only to a method, not out of a method. Because a Java array is an object, its reference is passed by value to a method, and changes to the array are visible to the caller of the method. Therefore, arrays must be used for output parameters in a Java method.

An output parameter accepts only one value in the first element of the array at index 0. Any attempt to return more than one value to an output parameter results in a Java exception.

For each output parameter, specify the Java type followed by empty square brackets (`[]`) to indicate that the type is an array. For example, specify an `int` type as `int[]` for an output parameter in the Java signature.

To return multiple values from a Java method, use an output parameter for each returned value. For example, the `supplierInfo()` method returns a supplier's name, address, city, state, and post code, each as a single string in an output parameter:

```
public static void supplierInfo(BigDecimal suppNum,
                                String[] suppName,
                                String[] streetAddr,
                                String[] cityName,
                                String[] stateName,
                                String[] postCode)
    throws SQLException
{
...
```

The `supplyQuantities()` method returns an average quantity, a minimum quantity, and a maximum quantity to separate output parameters of the integer type:

```
public static void supplyQuantities(int[] avgQty,
                                    int[] minQty,
                                    int[] maxQty)
    throws SQLException
{
...
```

For more information about the SPJ examples, see Appendix A: Sample SPJs.

Type Mapping of Output Parameters

When writing an SPJ method, consider how the output of the SPJ will be used in the calling application. For output parameters, the Java data type of the SPJ method must map to an SQL/MX data type. See Table 2 The SQL/MX data type must then map to a compatible data type in the calling application. See Chapter 5: Invoking SPJs in NonStop SQL/MX, for the client application programming interfaces (APIs) that support SPJs and for cross-references to the appropriate manuals for type mappings between NonStop SQL/MX and the API.

Stored Procedure Result Sets

SQL/MX supports SPJs that return stored procedure result sets. A stored procedure result set is a cursor that is left open after the SPJ method executes (that is, after the CALL statement executes successfully). After the CALL statement executes successfully, the calling application can issue requests to open and then retrieve multiple rows of data from the returned result sets.

An SPJ method returns an ordered collection of result sets to the calling application by executing SELECT statements and placing each returned ResultSet object into a one-element Java array of type java.sql.ResultSet[ ]. The java.sql.ResultSet[ ] array is part of the Java method's signature and is recognized by SQL/MX as a container for a single stored procedure result set.

Place the java.sql.ResultSet[ ] parameters after the other Java parameters, if any, in the Java signature. If you do not place the java.sql.ResultSet[ ] parameters after the other parameters in the signature, SQL/MX prevents you from creating an SPJ using that Java method.

NOTE: Stored procedures in Java that return Result Sets are supported on systems running J06.05 and later J-series RVUs and H06.16 and later H-series RVUs.

The following example shows the declaration of SPJ method orderSummary for the sample stored procedure SALES.ORDER_SUMMARY. This Java method can return a maximum of two result sets.

```
public static void orderSummary(java.lang.String onOrAfter,
                                long[] numOrders,
                                java.sql.ResultSet[] orders,
                                java.sql.ResultSet[] detail)
```

The following code fragment shows how the SALES.ORDER_SUMMARY SPJ method returns one of its result sets by executing a SELECT statement and assigning the acquired java.sql.ResultSet object to a java.sql.ResultSet[ ] output array.

```
// Open a result set for <order num, order info> rows
s = " SELECT  AMOUNTS.*, ORDERS.order_date, EMPS.last_name " +
" FROM       ( select o.ordernum, count(d.partnum)as num_parts, " +
"               sum(d.unit_price * d.qty_ordered) as amount     " +
"             from sales.orders o, sales.odetail d              " +
"             where o.ordernum = d.ordernum                     " +
"              and o.order_date >= cast(? as date)              " +
"             group by o.ordernum ) AMOUNTS,                    " +
"                sales.orders ORDERS, persnl.employee EMPS         " +
" WHERE          AMOUNTS.ordernum = ORDERS.ordernum                " +
" AND            ORDERS.salesrep = EMPS.empnum                     " +
" ORDER BY       ORDERS.ordernum                                    ";
PreparedStatement ps2 = c.prepareStatement(s);
ps2.setString(1, onOrAfter);
// Assign the returned result set object to the first element of a
// java.sql.ResultSet[] output array
orders[0] = ps2.executeQuery();
```

For the complete example, see the ORDERSUMMARY Stored Procedure (page 121).

△ **CAUTION:** In an SPJ method that returns result sets, do not explicitly close the default connection or the statement object. SQL/MX closes the connection used to return result sets after it finishes processing the result sets. If you close the connection on which the result sets are being returned, those result sets will be lost, and the calling application will not be able to process them.

Using the main() Method

You can use the `main()` method of a Java class file as an SPJ method. The `main()` method is different from other Java methods because it accepts input values in an array of `java.lang.String` objects and does not return any values in its array parameter.

For example, you can register this `main()` method as an SPJ:

```
public static void main (java.lang.String [] args) {
   ...
}
```

When you register a `main()` method as an SPJ, the CREATE PROCEDURE statement can contain zero or more SQL parameters, even though the underlying `main()` method has only one array parameter. All the SQL parameters of the SPJ must have the character string data type, CHAR or VARCHAR, and be declared with the IN mode.

If you specify the optional Java signature in the EXTERNAL NAME clause of the CREATE PROCEDURE statement, the signature must be `(java.lang.String [])`. For more information about creating an SPJ, see the Using the CREATE PROCEDURE Statement (page 59).

Null Input and Output

You can pass a null value as input to or output from an SPJ method, provided that the Java data type of the parameter supports nulls. Java primitive data types do not support nulls. However, Java wrapper classes that correspond to primitive data types do support nulls. If a null is input or output for a parameter that does not support nulls, NonStop SQL/MX raises an error condition.

To anticipate null input or output for your SPJ, use Java wrapper classes instead of primitive data types in the method signature.

For example, this Java method uses a Java primitive data type in its signature where no null values are expected:

```
public static void employeeJob(int empNum)
```

This Java method also uses a Java wrapper class in its signature to anticipate a possible returned null value:

```
public static void employeeJob(Integer[] jobCode)
```

Static Java Variables

To ensure that your SPJ method is portable, you should avoid declaring static variables in the method. NonStop SQL/MX does not ensure the scope and persistence of static Java variables.

Consider these drawbacks to using static variables in your SPJ method:

- If an SQL/MX UDR server crashes, the calling application receives a new SQL/MX UDR server and a new SPJ environment, including new copies of all static variables initialized in that server. For more information, see SQL/MX UDR Server Process (page 25).
- When application classes contain static variables, NonStop SQL/MX does not ensure that exactly one copy of those static variables is in a given SQL/MX UDR server process or SPJ environment. For more information, see Class Loaders in an SPJ Environment (page 28) .

Nested Java Method Invocations

An SPJ that invokes another SPJ by issuing a CALL statement causes an additional SQL/MX UDR server process to be created. Nesting SQL/MX UDR server processes in this way wastes resources and diminishes performance. If you want an SPJ method to call another SPJ method, invoke the other Java method directly through Java instead of using a CALL statement.

Accessing SQL/MP and SQL/MX Databases

SPJ methods that access an SQL/MP or SQL/MX database must be from a Java class that uses JDBC/MX method calls.

- Use of java.sql.Connection Objects (page 53)
- JDBC/MX-Based Java Method (page 54)
- Referring to Database Objects in an SPJ Method (page 55)
- Exception Handling (page 56)

Use of java.sql.Connection Objects

NonStop SQL/MX supports a default connection in an SPJ execution environment, which has a data source URL of `"jdbc:default:connection"`. For example:

```
Connection conn = DriverManager.getConnection("jdbc:default:connection");
```

`java.sql.Connection` objects that use the `"jdbc:default:connection"` URL are portable to the NonStop SQL/MX platform from other database management systems (DBMSs).

NonStop SQL/MX controls default connections in the SPJ environment and closes default connections when they are no longer needed. Therefore, you do not need to use the `close()` method in an SPJ method to explicitly close a default connection when the connection is no longer needed. In fact, if an SPJ method returns result sets, you should not explicitly close the default connection. NonStop SQL/MX closes the connection used to return result sets after it finishes processing the result sets. If an SPJ method closes the connection on which the result sets are being returned, those result sets will be lost, and the calling application will not be able to process them.

A default connection that is acquired when an SPJ method executes is not guaranteed to remain open for future invocations of the SPJ method. Therefore, do not store default connections in static variables for future use.

The default connection URL, `"jdbc:default:connection"`, is invalid outside a DBMS, such as when you execute a Java method in an application. To write an SPJ method that operates in a DBMS, in an application, or both, without having to change and recompile the code, use the `sqlj.defaultconnection` system property:

```
String s = System.property("sqlj.defaultconnection");
if (s == null) {
s = other-url;
}
Connection c = DriverManager.getConnection(s);
```

The value of `sqlj.defaultconnection` is `"jdbc:default:connection"` in a DBMS and null outside a DBMS.

The SPJ environment sets the initial connection pool size to 1, but it does not limit the number of connections an SPJ method can make. The SPJ environment also sets the minimum connection pool size to 1 so that there is always at least one connection available in the pool. The default settings in the SPJ environment are:

- `maxPoolSize=0`
- `minPoolSize=1`
- `initialPoolSize=1`

To change these settings, use the properties parameter of the `DriverManager.getConnection()` method as shown below:

```
java.util.Properties props = new Properties();
props.setProperty("maxPoolSize", "10");
props.setProperty("minPoolSize", "5");
props.setProperty("initialPoolSize", "5");
Connection conn =
DriverManager.getConnection("jdbc:default:connection", props);
```

For more information on JDBC Type 2 driver properties, see the *JDBC Type 2 Driver Programmer's Reference for SQL/MX Release 3.2*.

JDBC/MX-Based Java Method

A JDBC/MX-based Java method is from a Java program that contains SQL/MX statements in JDBC/MX method calls. You can use this type of method to create an SPJ that performs SQL operations on an SQL/MP or SQL/MX database.

For example, the `adjustSalary()` method in the `Payroll` class adjusts an employee's salary in the EMPLOYEE table:

```
public class Payroll {

   public static void adjustSalary(BigDecimal empNum,
                                   double percent,
                                   BigDecimal[] newSalary)
       throws SQLException
   {
       Connection conn = DriverManager.getConnection("jdbc:default:connection");

       PreparedStatement setSalary =
           conn.prepareStatement("UPDATE samdbcat.persnl.employee " +
                                 "SET salary = salary * (1 + (? / 100)) " +
                                 "WHERE empnum = ?");

       PreparedStatement getSalary =
           conn.prepareStatement("SELECT salary " +
                                 "FROM samdbcat.persnl.employee " +
                                 "WHERE empnum = ?");

       setSalary.setDouble(1, percent);
       setSalary.setBigDecimal(2, empNum);
       setSalary.executeUpdate();

       getSalary.setBigDecimal(1, empNum);
       ResultSet rs = getSalary.executeQuery();
       rs.next();
       newSalary[0] = rs.getBigDecimal(1);
       rs.close();

       conn.close();
   }
}
```

You do not have to explicitly load the JDBC/MX driver before establishing a connection to the database. The SQL/MX UDR server automatically loads the JDBC/MX driver when the SPJ is called.

To register this method as an SPJ in NonStop SQL/MX, use a CREATE PROCEDURE statement. For details, see Chapter 4: Registering SPJs in NonStop SQL/MX.

For other examples of JDBC/MX-based SPJs, see Appendix A: Sample SPJs.

For information about JDBC/MX, see the *JDBC Driver for SQL/MX Programmer's Reference*.

Referring to Database Objects in an SPJ Method

In an SPJ method, you can refer to both SQL/MP and SQL/MX database objects as you would in other SQL/MX applications. SQL/MX database objects have three-part ANSI names that include the catalog, schema, and object name. SQL/MP database objects have Guardian file names that include the node, volume, subvolume, and file name. Typically, SQL/MP aliases, which are three-part logical names, are used to refer to SQL/MP database objects. For more information about database object names, see the *SQL/MX Reference Manual*.

How you qualify three-part object names in an SPJ method depends on the SQL/MX release that you are using:

- Object Name Qualification Before NonStop SQL/MX Release 2.1.1 (page 55)
- Object Name Qualification in NonStop SQL/MX Release 2.1.1 and Later (page 55)

Object Name Qualification Before NonStop SQL/MX Release 2.1.1

Before SQL/MX Release 2.1.1 (ABX SPRs), the catalog and schema values of referenced database objects are not set in the SPJ environment. As a result, you must fully qualify database objects that are referenced in SPJ methods. This SPJ method uses the fully qualified object name, SAMDBCAT.SALES.ORDERS:

```
public static void numDailyOrders(Date date,
                                  int[] numOrders)
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getNumOrders =
        conn.prepareStatement("SELECT COUNT(order_date) " +
                              "FROM samdbcat.sales.orders " +
                              "WHERE order_date = ?");


    getNumOrders.setDate(1, date);
    ResultSet rs = getNumOrders.executeQuery();
    rs.next();
    numOrders[0] = rs.getInt(1);
    rs.close();

    conn.close();
}
```

Using fully qualified object names in SPJ methods makes the SPJ methods less portable from one system to another, where catalog and schema names might differ. However, if you do not fully qualify the database object names, the default catalog and schema values will be the same as those in the SYSTEM_DEFAULTS table.

Object Name Qualification in NonStop SQL/MX Release 2.1.1 and Later

In SQL/MX Release 2.1.1 and later, the SQL/MX UDR server propagates the values of the catalog and schema where the SPJ is registered to the SPJ environment. By default, database connections created in the SPJ method are associated with those catalog and schema values, meaning that partially qualified objects with one- or two-part names in the SPJ method are qualified with the same catalog and schema values as the SPJ. For example, this SPJ method, which is registered as an SPJ in the SAMDBCAT.SALES schema, refers to the unqualified database object, ORDERS:

```
public static void numDailyOrders(Date date,
                                  int[] numOrders)
    throws SQLException
```

```
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getNumOrders =
        conn.prepareStatement("SELECT COUNT(order_date) " +
                              "FROM orders " +
                              "WHERE order_date = ?");


    getNumOrders.setDate(1, date);
    ResultSet rs = getNumOrders.executeQuery();
    rs.next();
    numOrders[0] = rs.getInt(1);
    rs.close();

    conn.close();
}
```

In the SPJ environment, the ORDERS table is qualified by default with the same catalog and schema, SAMDBCAT.SALES, as the SPJ.

The default behavior takes effect only when `getConnection()` and UDR_JAVA_OPTIONS do not contain catalog or schema properties. Catalog and schema property values specified in UDR_JAVA_OPTIONS have higher precedence over the default behavior. Catalog and schema property values in `getConnection()` have higher precedence over both the default behavior and UDR_JAVA_OPTIONS.

To override the default catalog and schema values and associate a database connection in an SPJ method with a different catalog or schema, specify the catalog or schema properties during connection creation. For example, `getConnection()` in this SPJ method specifies the catalog, CAT, which overrides the default catalog, SAMDBCAT, while the default schema remains SALES:

```
public static void numDailyOrders(Date date,
                                  int[] numOrders)
    throws SQLException
{
    Properties prop = new Properties();
    prop.setProperty("catalog","CAT");

    Connection conn = DriverManager.getConnection("jdbc:default:connection", prop);

    PreparedStatement getNumOrders =
        conn.prepareStatement("SELECT COUNT(order_date) " +
                              "FROM orders " +
                              "WHERE order_date = ?");


    getNumOrders.setDate(1, date);
    ResultSet rs = getNumOrders.executeQuery();
    rs.next();
    numOrders[0] = rs.getInt(1);
    rs.close();

    conn.close();
}
```

Be aware that overriding the default values by using `getConnection()` or UDR_JAVA_OPTIONS requires you to hard-code the catalog and schema values and might make SPJ methods less portable across systems.

Exception Handling

For SPJ methods that access an SQL/MP or SQL/MX database, no special code is necessary for handling exceptions. If an SQL operation fails inside the SPJ, the error message associated with the failure is returned to the application that issues the CALL statement.

Handling Java Exceptions

If an SPJ method returns an uncaught Java exception or an uncaught chain of `java.sql.SQLException` objects, NonStop SQL/MX converts each Java exception object into an SQL/MX error condition, and the CALL statement fails. Each SQL/MX error condition contains the message text associated with one Java exception object.

If an SPJ method catches and handles exceptions itself, those exceptions do not affect SQL/MX processing.

User-Defined Exceptions

The SQLSTATE values 38001 to 38999 are reserved for you to define your own error conditions that SPJ methods can return. By coding your SPJ method to throw a `java.sql.SQLException` object, you cause the CALL statement to fail with a specific user-defined SQLSTATE value and your own error message text.

If you define the SQLSTATE to be outside the range of 38001 to 38999, NonStop SQL/MX raises SQLSTATE 39001, external routine invocation exception.

This example uses the `throw` statement in the SPJ method named `numMonthlyOrders()` to raise a user-defined error condition when an invalid argument value is entered for the month:

```
public static void numMonthlyOrders(int month,
                                    int[] numOrders)
   throws java.sql.SQLException
{
    if ( month < 1 || month > 12 )
    {
      throw new java.sql.SQLException (
          "Invalid value for month. "
        + "Retry the CALL statement using a number "
        + "from 1 to 12 to represent the month.", "38001" );
    }
    ....
}
```

For more information about the `numMonthlyOrders()` method, see the Sales Class (page 107).

For information about specific SQL/MX errors, see the *SQL/MX Messages Manual*, which lists the SQLCODE, SQLSTATE, message text, and cause-effect-recovery information for all SQL/MX errors.

Writing Data to a File or Terminal

Within the SPJ environment, data sent to the `System.out` and `System.err` output streams goes nowhere by default because the SQL/MX UDR server runs without a terminal. You can add code to your SPJ methods to write data or debugging information to an OSS file. You can also map one or both of the `System.out` and `System.err` streams to a file.

Because terminal devices have OSS path names, you can route output from an SPJ method to a terminal device if you know the OSS name of that device. The `tty` command in OSS writes the full path name of your terminal device to standard output.

This example enables an SPJ method to write a message into a file:

```
public static void adjustSalary(BigDecimal empNum,
                                double percent,
                                BigDecimal[] newSalary)
   throws SQLException
{
   ...
   String outputFileName = "/usr/mydir/spj.output";
   FileOutputStream fileStream =
       new FileOutputStream(outputFileName);
   PrintStream printStream =
       new PrintStream(fileStream, true);
   ...
   printStream.println("The salary was updated for employee "
                        + empnum);
   ...
}
```

Suppose that an employee number of 202 is passed to the SPJ. When a CALL statement invokes this SPJ method, the text `"The salary was updated for employee 202"` is written to the file `/usr/mydir/spj.output`.

You can also redirect one or both `System` streams to an output stream of your choice. This example extends the code from the previous example for stream redirection:

```
public static void adjustSalary(BigDecimal empNum,
                                double percent,
                                BigDecimal[] newSalary)
   throws SQLException
{
   ...
   String outputFileName = "/usr/mydir/spj.output";
   FileOutputStream fileStream =
       new FileOutputStream(outputFileName);
   PrintStream printStream =
       new PrintStream(fileStream, true);
   System.setOut(printStream);
   System.setErr(printStream);
   ...


  // This message goes to the output stream.
   System.out.println("The salary was updated for employee "
                        + empnum);
  // This message goes to the error stream.
   System.err.println("The salary was not updated for employee "
                        + empnum);
   ...
}
```

When a CALL statement invokes the SPJ method, a message is written to the file `/usr/mydir/spj.output`. If the invocation succeeds, this message is written to the file:

```
The salary was updated for employee 202
```

If the invocation fails, this message is written to the file:

```
The salary was not updated for employee 202
```

Compiling Java Classes

Before registering a Java method as an SPJ, you must compile your Java source file into Java bytecode by using the Java programming language compiler (`javac`). To compile a class file, see the *NonStop Server for Java Programmer's Reference* or the *NonStop Server for Java Tools Reference Pages*.

4 Registering SPJs in NonStop SQL/MX

This section covers these topics:

This section assumes that you have already written and compiled the SPJ methods. For more information, see Chapter 3: Writing SPJ Methods.

Creating an SPJ

The CREATE PROCEDURE statement registers an existing Java method as an SPJ within an SQL/MX database. Registration of an SPJ enables any SQL/MX application or interface to execute the SPJ method by using a CALL statement.

You can issue a CREATE PROCEDURE statement from any application or interface that calls NonStop SQL/MX, such as:

- SQL/MX conversational interface (MXCI)
- Embedded SQL programs in C, C++, or COBOL
- JDBC/MX programs
- NonStop ODBC/MX clients
- NSM/web

NOTE: The catalog and schema of the SPJ must exist when you run the CREATE PROCEDURE statement.

Required Privileges for Creating an SPJ

To issue a CREATE PROCEDURE statement, you must be the owner of the schema, or the super ID (that is, super.super), and have read access to the Java class file or JAR file that contains the SPJ method.

Effect of Creating an SPJ

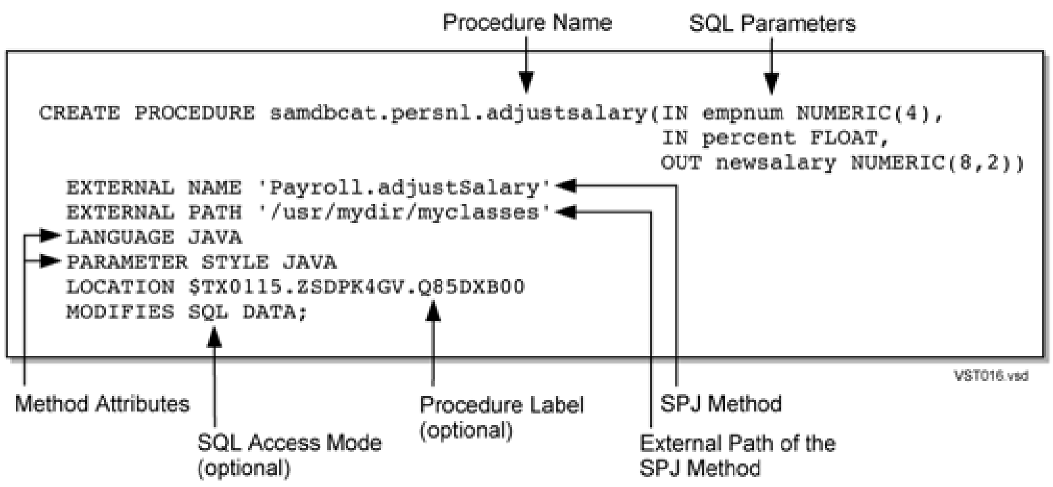
When you execute a CREATE PROCEDURE statement, the identified Java method is verified to exist in the Java class file and to have attributes that are consistent with those specified in the CREATE PROCEDURE statement.

Successful execution of the CREATE PROCEDURE statement updates rows in the system metadata tables and creates aprocedure label for the SPJ. If the CREATE PROCEDURE statement fails to execute, NonStop SQL/MX does not update system metadata tables and does not create a procedure label.

For more information, see the Effect of Registering an SPJ (page 21).

Using the CREATE PROCEDURE Statement

This subsection explains how to write a CREATE PROCEDURE statement, as shown in this example:


Procedure Name
SQL Parameters
Method Attributes
SQL Access Mode (optional)
Procedure Label (optional)
SPJ Method
External Path of the SPJ Method
VST016.vsd

```
CREATE PROCEDURE samdbcat.persnl.adjustsalary(IN empnum NUMERIC(4),
                                              IN percent FLOAT,
                                              OUT newsalary NUMERIC(8,2))
  EXTERNAL NAME 'Payroll.adjustSalary'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  LOCATION $TX0115.ZSDPK4GV.Q85DXB00
  MODIFIES SQL DATA;
```

To construct a CREATE PROCEDURE statement, see:

- Naming the Stored Procedure (page 60)
- Specifying SQL Parameters (page 61)
- Specifying the SPJ Method (page 64)
- Specifying the External Path (page 66)
- Naming the Procedure Label (page 66)
- Specifying an SQL Access Mode (page 67)

The examples in this subsection are based on the SPJ methods in Appendix A: Sample SPJs.

For the syntax of the CREATE PROCEDURE statement, see the *SQL/MX Reference Manual*.

Naming the Stored Procedure

Choose the name of the procedure and specify it in the form of an ANSI logical name, as shown:

```
CREATE PROCEDURE samdbcat.persnl.adjustsalary ...
```

The procedure name you select is used in the CALL statements that invoke the SPJ and identifies the underlying SPJ method.

When selecting a procedure name:

- Choose a meaningful and unique name that does not already exist for a procedure, table, view, or SQL/MP alias in the same schema.

  For example, use the procedure name ADJUSTSALARY for a Java method named `adjustSalary()` that adjusts an employee's salary in the EMPLOYEE table.

- Use delimited identifiers sparingly. The procedure name is upshifted for interpretation except for the parts that you specify as delimited identifiers.

  For example, if you specify a procedure name of `AdjustSalary` in the CREATE PROCEDURE statement, the interpreted name is `ADJUSTSALARY`. To call this SPJ, you can use lowercase or uppercase for the procedure name, as shown:

  ```
  CALL samdbcat.persnl.adjustsalary(202, 5.5, ?);
  ```

  If you specify a procedure name of `"AdjustSalary"` in the CREATE PROCEDURE statement, the interpreted name is `"AdjustSalary"`. To call this SPJ, you must delimit the case-sensitive part of the procedure name in double quotes, as shown:

  ```
  CALL samdbcat.persnl."AdjustSalary"(202, 5.5, ?);
  ```

  For information about delimited identifiers, see the *SQL/MX Reference Manual*.

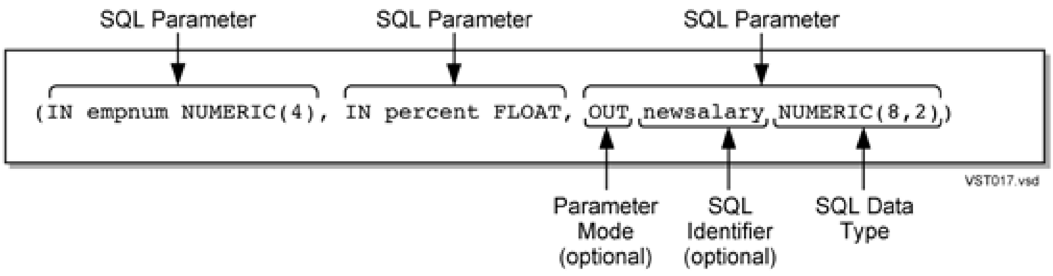
- Qualify the procedure name with an existing catalog and schema. If you do not fully qualify the procedure name, NonStop SQL/MX qualifies it according to the current settings of CATALOG and SCHEMA in the SYSTEM_DEFAULTS table. If the NAMETYPE attribute is set to NSK instead of ANSI and you do not fully qualify the procedure name, NonStop SQL/MX returns an error. For more information, see the *SQL/MX Reference Manual*.
- Note that NonStop SQL/MX does not support the overloading of procedure names and disallows the use of the same procedure name with different signatures. That is, you cannot register the same procedure name more than once with different underlying SPJ methods.

Specifying SQL Parameters

If the SPJ method does not accept arguments, like the `lowerPrice()` method, specify the procedure name with empty parentheses, as shown:

```
samdbcat.sales.lowerprice()
```

If the SPJ method accepts arguments, specify the corresponding SQL parameters. Each SQL parameter consists of a parameter mode, an optional SQL identifier, and an SQL data type, as shown:


SQL Parameter
SQL Parameter
SQL Parameter
(IN empnum NUMERIC(4), IN percent FLOAT, OUT newsalary NUMERIC(8,2))
VST017.vsd
Parameter Mode (optional)
SQL Identifier (optional)
SQL Data Type

Parameter Mode

The parameter mode specifies the input and output mode of an SQL parameter:

- IN specifies a parameter that passes a single value to an SPJ.
- INOUT specifies a parameter that passes a single value to and accepts a single value from an SPJ.
- OUT specifies a parameter that accepts a single value from an SPJ.

The default mode is IN if you do not specify the parameter mode. For a parameter mode of OUT or INOUT, you must explicitly specify the parameter mode for the SQL parameter.

In this example, all three parameters pass data to the SPJ, and the last parameter accepts data from the SPJ:

```
samdbcat.sales.totalprice(IN qty NUMERIC(18),
                          IN rate VARCHAR(10),
                          INOUT price NUMERIC(18,2))
```

SQL Identifier

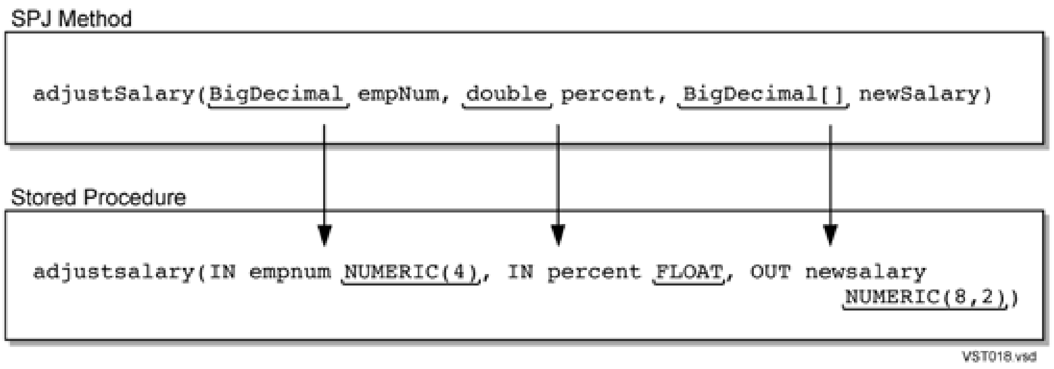
The SQL identifier is an optional name you can use to describe each SQL parameter and to enhance the readability of the CREATE PROCEDURE statement. For example, these identifiers describe the SQL parameters of the ADJUSTSALARY procedure:

```
samdbcat.persnl.adjustsalary(IN empnum NUMERIC(4),
                             IN percent FLOAT,
                            OUT newsalary NUMERIC(8,2))
```

The identifier conforms to the syntax of other SQL identifiers. For information about identifiers, see the *SQL/MX Reference Manual*.

SQL Data Type

The SQL data types must correspond with the underlying Java data types of the SPJ method, as shown:


SPJ Method
adjustSalary(BigDecimal empNum, double percent, BigDecimal[] newSalary)
Stored Procedure
adjustsalary(IN empnum NUMERIC(4), IN percent FLOAT, OUT newsalary NUMERIC(8,2))
VST018.vsd

Because the Java signature of the SPJ method in this example is `(BigDecimal, double, BigDecimal[])`, you should specify these SQL data types, or similar ones, for the procedure: `(NUMERIC, FLOAT, NUMERIC)`. Recall that all output parameters (OUT and INOUT) of a Java method must be arrays that accept a single value. (See Returning Output Values From the Java Method (page 50).) In this example, the type of the output parameter, `BigDecimal[]`, corresponds with the SQL type, `NUMERIC`.

For mappings between SQL/MX and Java data types, see Table 3 (page 62)

Table 3 Mapping of SQL/MX Data Types to Java Data Types

SQL/MX Data Type	Maps to Java Data Type...
CHAR[ACTER]* CHAR[ACTER] VARYING* VARCHAR* Ext * PIC[TURE] X* NCHAR NCHAR VARYING NATIONAL CHAR[ACTER] NATIONAL CHAR[ACTER] VARYING	java.lang.String
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
NUMERIC** DEC[IMAL]** Ext PIC[TURE] S9	java.math.BigDecimal
SMALLINT**	short
INT[EGER]**	int (or java.lang.Integer if specified)***
LARGEINT	long (or java.lang.Long if specified)***
FLOAT	double (or java.lang.Double if specified)***
REAL	float (or java.lang.Float if specified)***

Table 3 Mapping of SQL/MX Data Types to Java Data Types *(continued)*

SQL/MX Data Type	Maps to Java Data Type...
`DOUBLE PRECISION`	`double` (or `java.lang.Double` if specified)***

* The character set for character string data types can be ISO88591 or UCS2.
** Numeric data types must be SIGNED, which is the default in NonStop SQL/MX.

*** By default, the SQL/MX data type maps to a Java primitive data type. The SQL/MX data type maps to a Java wrapper class only if you specify the wrapper class in the Java signature of the EXTERNAL NAME clause. See the Java Method Signature (page 64).

Ext This icon indicates an SQL data type that is an SQL/MX extension to the ANSI standard. All other SQL data types in this table conform to the ANSI standard.

For information about SQL/MX data types, see the *SQL/MX Reference Manual*. For cross-references to appropriate manuals for mappings of SQL/MX data types to compatible data types in calling applications, see Chapter 5: Invoking SPJs in NonStop SQL/MX.

Character String Parameters and Character Sets

For SQL parameters that have character string data types, you can specify either ISO88591 or UCS2 (Unicode) for the character set. This example uses UCS2 as the character set of the SQL parameter named `lastname`:

```
(IN empnum INT, OUT lastname CHAR(25) CHARACTER SET UCS2)
```

If you do not specify a character set, the default is ISO88591:

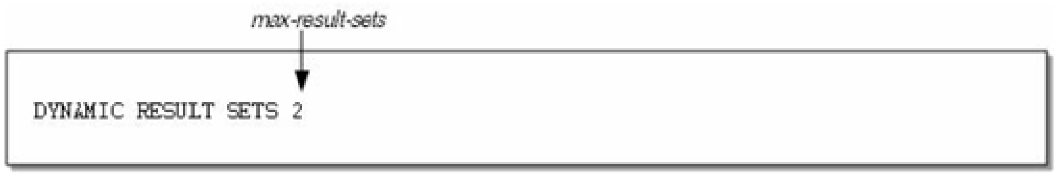
```
(IN empnum INT, OUT lastname CHAR(25))
```

If the SPJ method passes character string values to or from an SQL/MP table that has a KANJI (Japanese) or KSC5601 (Korean) column, use the ISO88591 character set for the SQL parameter.

For more information, see character sets in the *SQL/MX Reference Manual*.

Specifying the Maximum Number of Result Sets

You can specify the maximum number of result sets that the SPJ can return using the following syntax:



```
DYNAMIC RESULT SETS 2
```

SPJ RS supports values of max - result - sets from zero up to and including 255. If the DYNAMIC RESULT SETS clause is omitted, DYNAMIC RESULT SETS 0 is implicit.

NOTE: Note The values ranging from zero to 255 are supported only on systems running J06.05 and later J-series RVUs or on systems running H06.16 and later H-series RVUs. On systems running J06.04 and earlier J-series RVUs or H06.15 and earlier H-series RVUs, SQL/MX throws an error if the value for max-result-sets is not zero.

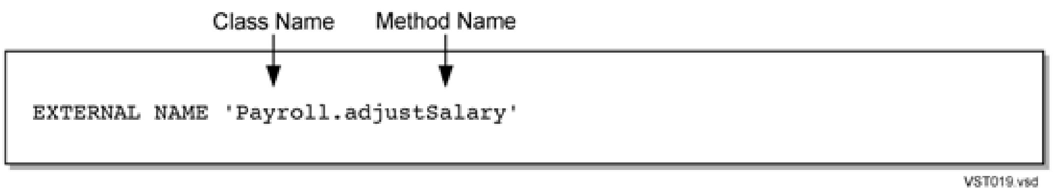
The actual number of result sets returned from a CALL statement and the column structure for those result sets are not known to SQL/MX until the stored procedure body executes.

When max - result - sets is a positive value, the Java method named in the CREATE PROCEDURE statement must have one or more "not necessarily max-result-sets" trailing parameters of type java.sql.ResultSet[ ]. The trailing Java parameters for result sets must follow the Java parameters (if any) associated with the procedure's declared SQL parameters.

If more than one matching Java method exists, by default SQL/MX does not attempt to resolve the ambiguity and an error is raised. This can happen when two or more methods in the specified Java class have the same overloaded method name but differ only in the number of trailing java.sql.ResultSet[ ] parameters. When a CREATE PROCEDURE statement can potentially map to more than one Java method, users must provide a full Java method signature in the EXTERNAL NAME clause to resolve the ambiguity.

Specifying the SPJ Method

Specify the external Java method to be used as the SPJ method in the EXTERNAL NAME clause, as shown:



The Java method that you specify in the EXTERNAL NAME clause must be defined as `public` and `static` and have a return type of `void`. For more information, see Guidelines for Writing SPJ Methods (page 49).

NonStop SQL/MX does not allow JAR file names to be specified in the EXTERNAL NAME clause. Instead, use the EXTERNAL PATH clause to specify the location of a JAR file that contains the Java class. See Specifying the External Path (page 66).

CAUTION: NonStop SQL/MX does not maintain the Java class file that contains the SPJ method. NonStop SQL/MX records only a reference to the location of the Java class file in system metadata. Subsequent changes to the Java class file after you register the SPJ might result in run-time errors when you try to execute the SPJ. For guidelines on how to alter an SPJ, see Altering an SPJ and Its Java Class (page 68).

Java Method Name

The Java method name consists of the case-sensitive names of the method and the Java class that contains the method. For example, the Java method named `adjustSalary()` is prefixed with the name of its class, `Payroll`:

```
'Payroll.adjustSalary'
```

If the class is stored in a package, you must also specify the package name. In this example, the package and subpackage names of the class file are `pkg.subpkg`:

```
'pkg.subpkg. Payroll.adjustSalary'
```

If you do not specify the class name and package, if it exists, in the EXTERNAL NAME clause, the CREATE PROCEDURE statement fails to register the SPJ.

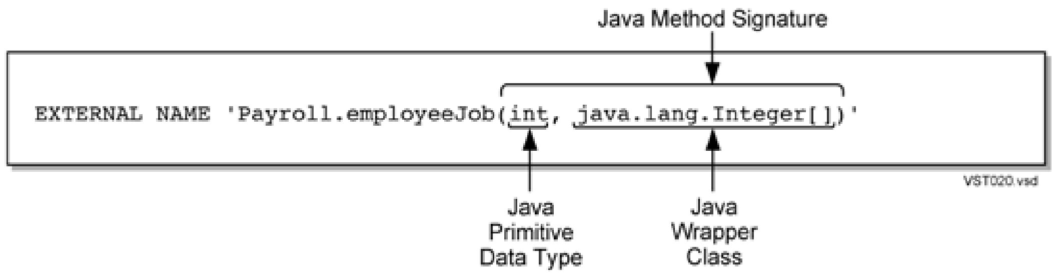
Java Method Signature

Specifying the Java signature is necessary only when an SQL parameter of the SPJ does not map by default to a Java wrapper class and when the SPJ method uses a Java wrapper class instead of a Java primitive data type in its Java signature. See Table 3: Mapping of SQL/MX Data Types to Java Data Types (page 62)

Usually, if you do not specify the Java signature, the SQL data types map by default to the correct Java data types. For example, this EXTERNAL NAME clause omits the Java signature because the SQL parameters (`NUMERIC` and `FLOAT`) map by default to the same Java data types (`java.math.BigDecimal` and `double`) as the parameters of the underlying Java method:

```
EXTERNAL NAME 'Payroll.adjustSalary'
```

However, if you specify `INT` for an SQL parameter and the underlying SPJ method uses `java.lang.Integer`, you must specify the Java signature in the EXTERNAL NAME clause, as shown:



The Java signature is case-sensitive and must be placed within parentheses. The signature must specify each of the parameter data types in the order that they appear in the Java method definition within the class file. Each Java data type that corresponds to an OUT or INOUT parameter must be followed by empty square brackets ([ ]) to indicate that it is an array parameter:

```
CREATE PROCEDURE samdbcat.persnl.employeejob(IN empnum INT,
                                             OUT jobcode INT)
     EXTERNAL NAME 'Payroll.employeeJob(int,
                                        java.lang.Integer [])'
```

An SPJ returns only one value to the first element of the array at index 0. An SPJ cannot return an array of values to an OUT or INOUT parameter. For more information, see Returning Output Values From the Java Method (page 50).

If you specify a Java signature string, NonStop SQL/MX verifies that the signature is a valid mapping of the parameters of the specified method into Java data types. If the signature is invalid, NonStop SQL/MX returns an error.

Even if you do not specify a Java signature string, NonStop SQL/MX generates a compressed representation of the Java method signature and stores it in the TEXT metadata table. The compressed signature conforms to the internal type signature emitted by the `javap` tool and the `-s` option (for example, `javap -s`*`classfile`*). NonStop SQL/MX returns an error if the compressed signature exceeds 8192 characters. For more information about `javap`, see the *NonStop Server for Java Tools Reference Pages*.

For example, consider the SPJ method, `adjustSalary()`:

```
public static void adjustSalary(BigDecimal empNum,
                                double percent,
                                BigDecimal[] newSalary)
...
```

The Java signature of this method in the CREATE PROCEDURE statement, if you choose to specify it, is:

```
(java.math.BigDecimal, double, java.math.BigDecimal[])
```

After you register the SPJ (that is, issue the CREATE PROCEDURE statement), you can display the compressed Java signature in the TEXT metadata table. Issue this query:

```
SELECT SUBSTRING(TEXT,1,100) AS "Compressed Signature"
FROM samdbcat.definition_schema_version_vernum.text t,
     nonstop_sqlmx_node.system_schema.all_uids u,
     nonstop_sqlmx_node.system_schema.schemata s
WHERE t.object_uid = u.object_uid
  AND u.object_name = 'ADJUSTSALARY'
  AND u.schema_uid = s.schema_uid
  AND s.schema_name = 'PERSNL';
```

The schema version number *`vernum`* for NonStop SQL/MX Releases 2.0, 2.1, and 2.2 is 1200.

The query displays the compressed Java signature of the ADJUSTSALARY procedure:

```
Compressed Signature
--------------------------------------------------------------------------------

(Ljava/math/BigDecimal;D[Ljava/math/BigDecimal;)V
```

For more information about the TEXT metadata table, see the *SQL/MX Reference Manual*.

Specifying the External Path

The external path specifies the location of the Java class file that contains the SPJ method. The external path is a case-sensitive string identifying the OSS directory or JAR file path where the Java class file (for example, `Payroll.class`) resides.

Package Consideration

Do not specify the package name in the EXTERNAL PATH clause. Instead, specify the package name in the EXTERNAL NAME clause. See Java Method Name (page 64).

Example of a Class File Path

For example, consider the `/usr/mydir/myclasses/Payroll.java` source file:

```
package pkg.subpkg;

public class Payroll
{
   public static void adjustSalary( BigDecimal empNum,
                                    double percent,
                                    BigDecimal[] newSalary )
   ...
```

The Java compiler generates this class file from the Java source file:

```
/usr/mydir/myclasses/pkg/subpkg/Payroll.class
```

The correct CREATE PROCEDURE clauses that refer to the `adjustSalary()` method in the class file are:

```
EXTERNAL NAME 'pkg.subpkg.Payroll.adjustSalary'
EXTERNAL PATH '/usr/mydir/myclasses'
```

Example of a JAR File Path

Suppose that the `Payroll.java` source file is compiled, and the class file is packaged in a JAR file, `myJar.jar`, which resides in the `/usr/mydir` directory:

```
/usr/mydir/myJar.jar
```

The correct CREATE PROCEDURE clauses that refer to the `adjustSalary()` method in the JAR file are:

```
EXTERNAL NAME 'pkg.subpkg.Payroll.adjustSalary'
EXTERNAL PATH '/usr/mydir/myJar.jar'
```

Naming the Procedure Label

The procedure label is used internally by NonStop SQL/MX to track privileges on an SPJ. Because the procedure label is used internally by NonStop SQL/MX, it is unnecessary to specify a Guardian name and location for it.

If you do not specify the LOCATION clause, NonStop SQL/MX generates a system-defined file with the volume name as specified in the DDL_DEFAULT_LOCATIONS default and subvolume as the schema of the SPJ, as shown:

```
\KINGPIN.$TX0115.ZSDX7KT4.SL9FSB00
```

If DDL_DEFAULT_LOCATIONS default is not set, then the volume name of the SPJ is retrieved from the =_DEFAULTS define.

If you want to control the location of the procedure label, particularly in a distributed database environment, specify the LOCATION clause, as shown:

```
LOCATION \REMOTENODE.$DISKVOL.SUBVOL.PROCLABEL
```

You can create the procedure label only on a node where the catalog of the SPJ is visible. For information about registering a catalog on a remote node, see the *SQL/MX Installation and Upgrade Guide and*SQL/MX Management Manual.

You can also use the LOCATION clause to specify only the volume of the procedure label:

```
LOCATION $DISKVOL
```

In this case, the node is the same node where the SPJ is registered, the subvolume is the same as the schema of the SPJ, and the file name is system generated.

For information about naming rules for the LOCATION clause, see the *SQL/MX Reference Manual*.

Specifying an SQL Access Mode

The SQL access mode indicates whether an SPJ performs SQL operations. Specifying the SQL access mode for an SPJ is entirely optional. If you do not specify the access mode in the CREATE PROCEDURE statement, the default is CONTAINS SQL, and the capability of the SPJ is MODIFIES SQL DATA.

If the SPJ does not perform SQL operations, specify the optional NO SQL clause. Even if the SPJ method does not perform SQL operations and the access mode is CONTAINS SQL, the SPJ still executes successfully. However, if you specify NO SQL for an SPJ method that performs SQL operations, NonStop SQL/MX returns an error when you try to invoke the SPJ.

To describe the types of SQL operations that an SPJ performs, specify one of these optional clauses: CONTAINS SQL, MODIFIES SQL DATA, or READS SQL DATA. All these options have the same capabilities as MODIFIES SQL DATA. Specify one of these options when registering a method that contains SQL statements and that is part of the class file of a JDBC/MX program. For information about JDBC/MX, see the *JDBC Driver for SQL/MX Programmer's Reference*.

Dropping an SPJ

The DROP PROCEDURE statement removes an SPJ from NonStop SQL/MX. You can issue a DROP PROCEDURE statement from any application or interface that calls NonStop SQL/MX, such as:

- SQL/MX conversational interface (MXCI)
- Embedded SQL programs in C, C++ or COBOL
- JDBC/MX programs
- NonStop ODBC/MX clients
- NSM/web

Required Privileges for Dropping an SPJ

To issue a DROP PROCEDURE statement, you must own the SPJ or be the super ID. To determine the ownership of an SPJ, see Showing Privileges on the SPJs (page 89).

Effect of Dropping an SPJ

Successful execution of the DROP PROCEDURE statement removes the SPJ attributes from system metadata tables and drops the procedure label of the SPJ. The underlying Java method and any dependent SPJ that refers to this SPJ are not affected by the DROP PROCEDURE statement. If the DROP PROCEDURE statement fails to execute, NonStop SQL/MX does not modify system metadata tables and does not drop the procedure label.

Using the DROP PROCEDURE Statement

To drop an SPJ, specify the name of the SPJ in the DROP PROCEDURE statement, as shown:

```
DROP PROCEDURE samdbcat.persnl.adjustsalary;
```

Do not specify the SQL parameters along with the procedure name. Each procedure name represents a unique SPJ in the database because NonStop SQL/MX does not support the overloading of procedure names.

For the syntax of the DROP PROCEDURE statement, see the *SQL/MX Reference Manual*.

Altering an SPJ and Its Java Class

To alter an SPJ, you must first drop the SPJ from system metadata by using the DROP PROCEDURE statement and then re-create the SPJ by using the CREATE PROCEDURE statement.

NOTE: NonStop SQL/MX does not support the ALTER_JAVA_PATH procedure from SQL/JRT of the ANSI SQL/Foundation standard.

To change the SPJ or the Java class of the SPJ method:

1. End all current SQL/MX UDR server processes that might have loaded the SPJ method that you want to change.

   For example, if you were calling the SPJ in an MXCI session, end that MXCI session. If you were calling the SPJ from an SQL/MX application, terminate that application.
2. Drop the SPJ from system metadata by using the DROP PROCEDURE statement. See Dropping an SPJ (page 67).

   NOTE: 2 is required only if you move the Java class of the SPJ method or change its external name or Java signature.
3. If desired, move the Java class file to another OSS directory, or change the source code of the SPJ method and recompile the Java class.
4. Re-create the SPJ by using the CREATE PROCEDURE statement. See Creating an SPJ (page 59).

   NOTE: 4 is required only if you move the Java class of the SPJ method or change its external name or Java signature.
5. Restart the application and call the SPJ in a new SQL/MX UDR server process.

△ **CAUTION:** To prevent unpredictable and undesirable behavior of CALL statements, do not change SPJ classes or referenced application classes while SPJ environments are active. For more information, see Maintaining Class and JAR Files in an SPJ Environment (page 30).

Typical scenarios that require an SPJ to be altered are discussed next:

- Changes to the External Path (page 68)
- Changes to the External Name (page 69)
- Changes to the Java Signature (page 70)

Changes to the External Path

The external path is the OSS directory or JAR file path of the Java class file that contains the SPJ method. If you change the external path of the underlying SPJ method, you must drop the associated SPJ and re-create it.

For example, suppose that you move the class, `Sales.class`, which contains the SPJ method, `numMonthlyOrders()`, from the `/usr/mydir/myclasses` directory into a JAR file, `myJar.jar`.

The SPJ was originally registered in system metadata as:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                              OUT number INT)
   EXTERNAL NAME 'pkg.subpkg.Sales.numMonthlyOrders'
   EXTERNAL PATH '/usr/mydir/myclasses'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   READS SQL DATA;
```

To drop the SPJ, issue this DROP PROCEDURE statement:

```
DROP PROCEDURE samdbcat.sales.monthlyorders;
```

To re-create the SPJ, issue this CREATE PROCEDURE statement:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                              OUT number INT)
   EXTERNAL NAME 'pkg.subpkg.Sales.numMonthlyOrders'
   EXTERNAL PATH '/usr/myapps/myJar.jar'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   READS SQL DATA;
```

If a statically compiled program calls this SPJ, you should explicitly recompile the module of that application. If you do not, NonStop SQL/MX tries to automatically recompile the SQL plan of the CALL statement at run time.

You can now successfully execute the relocated SPJ method by issuing the existing CALL statements.

Changes to the External Name

The external name identifies the SPJ method that is registered as an SPJ. A change to the external name of an SPJ suggests either a reassignment of the SPJ method or a fundamental change to the underlying Java class of the existing SPJ method.

For example, suppose that you reassign the SPJ named MONTHLYORDERS from the SPJ method `numMonthlyOrders()` to another SPJ method named `numMonthlyOrders2()`, which has the same Java signature.

The SPJ named MONTHLYORDERS was originally registered in system metadata as:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                              OUT number INT)
   EXTERNAL NAME 'pkg.subpkg.Sales.numMonthlyOrders'
   EXTERNAL PATH '/usr/myapps/myJar.jar'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   READS SQL DATA;
```

To drop the SPJ, issue this DROP PROCEDURE statement:

```
DROP PROCEDURE samdbcat.sales.monthlyorders;
```

To re-create the SPJ, issue this CREATE PROCEDURE statement:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                              OUT number INT)
   EXTERNAL NAME 'pkg.subpkg.Sales.numMonthlyOrders2'
   EXTERNAL PATH '/usr/myapps/myJar.jar'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   READS SQL DATA;
```

If a statically compiled program calls this SPJ, you should explicitly recompile the module of that application. If you do not, NonStop SQL/MX tries to automatically recompile the SQL plan of the CALL statement at run time.

You can now successfully execute the new SPJ method by issuing the existing CALL statements.

Changes to the Java Signature

The Java signature specifies each of the parameter data types in the Java method definition. A change to the Java signature of an SPJ suggests a fundamental change to the underlying Java class of the existing SPJ method.

For example, suppose that you reassign the SPJ named MONTHLYORDERS from the SPJ method named `numMonthlyOrders()` to another SPJ method named `numMonthlyOrders2()`, which has a different Java signature.

The SPJ named MONTHLYORDERS was originally registered in system metadata as:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                              OUT number INT)
   EXTERNAL NAME 'pkg.subpkg.Sales.numMonthlyOrders'
   EXTERNAL PATH '/usr/myapps/myJar.jar'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   READS SQL DATA;
```

To drop the SPJ, issue this DROP PROCEDURE statement:

```
DROP PROCEDURE samdbcat.sales.monthlyorders;
```

To re-create the SPJ, issue this CREATE PROCEDURE statement:

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT,
                                          IN partnum NUMERIC(4),
                                              OUT number INT)
   EXTERNAL NAME 'pkg.subpkg.Sales.numMonthlyOrders2'
   EXTERNAL PATH '/usr/myapps/myJar.jar'
   LANGUAGE JAVA
   PARAMETER STYLE JAVA
   READS SQL DATA;
```

You must update the CALL statements in the source code of the calling applications so that the arguments match the new parameters of the SPJ. If the application is statically compiled, you must recompile the entire application. You can now successfully execute the new SPJ method by issuing the updated CALL statements.

5 Invoking SPJs in NonStop SQL/MX

This section describes how to execute SPJs and assumes that you have already registered the SPJs in NonStop SQL/MX. For information about registering an SPJ, see Chapter 4: Registering SPJs in NonStop SQL/MX.

This section covers these topics:

Calling an SPJ

The CALL statement invokes an SPJ in NonStop SQL/MX. You can issue the CALL statement from any of these applications or interfaces that call NonStop SQL/MX:

- SQL/MX conversational interface (MXCI)
- JDBC/MX programs
- Embedded SQL programs in C, C++, or COBOL
- NonStop ODBC/MX clients

You can use the CALL statement only as a stand-alone SQL statement. You cannot use the CALL statement inside a compound statement or with rowsets.

Required Privileges for Calling an SPJ

To execute the CALL statement, you must have the EXECUTE privilege on the procedure. For more information, see Granting Privileges for Invoking SPJs (page 83).

Effect of Calling an SPJ

When you issue a CALL statement, the SPJ method of the invoked SPJ executes inside a JVM within an SPJ environment. For more information, see the Effect of Invoking an SPJ (page 22).

△ **CAUTION:** NonStop SQL/MX does not maintain the Java class file that contains the SPJ method. NonStop SQL/MX records only a reference to the location of the Java class file in system metadata. Subsequent changes to the Java class file after you register an SPJ might result in run-time errors when you try to execute it. For guidelines on how to alter an SPJ, see Altering an SPJ and Its Java Class (page 68).

Transaction Behavior

A CALL statement automatically initiates a transaction if there is no active transaction. An SPJ method invoked by a CALL statement usually inherits the transaction from its caller, except when `jdbcmx.transactonMode` is set to `internal`. For more information, see the Effect of the jdbcmx.transactionMode Property (page 72).

Transaction Statements or Methods in an SPJ Method

You can start transactions within a stored procedure by performing the following steps:

1. Suspend the current transaction (if supported by the API).
2. Start a new transaction.
3. Complete the work.
4. End the new transaction.
5. Resume the current transaction (if suspended).

You must ensure that the transactions started within the stored procedure are cleaned up properly. Some of the Java APIs do not support the ability to suspend and resume transactions. In this case, TMF marks the newly started transaction as the current transaction even though original transaction is still active, because an SPJ will be executed in a multi-threaded environment.

NOTE: SPJs that return data via result sets are not allowed to use transactions started within an SPJ for statements that populate the result set.

Committing or Rolling Back a Transaction

In an application, the failure of a CALL statement does not always automatically abort a transaction:

- When AUTOCOMMIT is ON during a system-initiated transaction, errors that occur during the execution of a CALL statement cause NonStop SQL/MX to roll back the transaction automatically, including database operations performed by the SPJ method, at the end of statement execution.
- When AUTOCOMMIT is OFF during a system-initiated or user-defined transaction, errors that occur during the execution of a CALL statement do not cause NonStop SQL/MX to roll back the transaction automatically.

To ensure an atomic unit of work in a user-defined transaction or when autocommit is OFF during a system-defined transaction, you should explicitly commit the transaction in the calling application when the transaction is successful and roll back the transaction in the calling application when an error occurs. For more information about transaction management, see the *SQL/MX Reference Manual*.

Effect of the jdbcmx.transactionMode Property

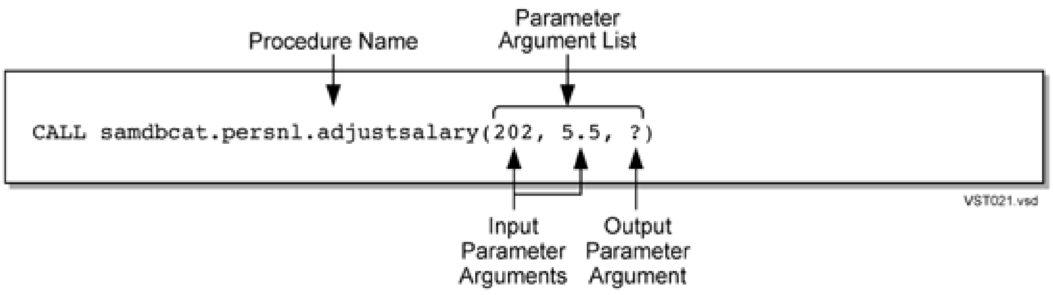
If you set the `jdbcmx.transactionMode` property in a UDR_JAVA_OPTIONS attribute, you might affect the transaction behavior of an SPJ. By default, an SPJ method runs in the default `mixed` transaction mode. The `mixed` and `external` transaction modes do not interfere with transaction behavior of the calling application.

However, if you set `jdbcmx.transactionMode` to `internal` in a UDR_JAVA_OPTIONS attribute, the external transaction of the calling application will be suspended, and the SQL operations of the SPJ method will be executed in an internal JDBC-managed transaction. The `internal` transaction mode prevents an SPJ method from inheriting a transaction from its caller and prevents the calling application from determining the outcome of an internal JDBC-managed transaction. For those reasons, do not use the `internal` transaction mode in an SPJ environment.

For information about `jdbcmx.transactionMode`, see the *JDBC Driver for SQL/MX Programmer's Reference*.

Using the CALL Statement

To invoke a stored procedure, specify the name of the stored procedure and its arguments in a CALL statement, as shown:


Procedure Name
Parameter Argument List
CALL samdbcat.persnl.adjustsalary(202, 5.5, ?)
Input Parameter Arguments
Output Parameter Argument
VST021.vsd

For the syntax of the CALL statement, see the *SQL/MX Reference Manual*.

Specifying the Name of the SPJ

In the CALL statement, specify the name of an SPJ that you have already registered in NonStop SQL/MX. Qualify the procedure name with the same catalog and schema that you specified in the CREATE PROCEDURE statement. For example:

```
SET SCHEMA samdbcat.persnl;
CALL adjustsalary(202, 5.5, ?);
```

If you do not fully qualify the procedure name, NonStop SQL/MX qualifies it according to the current settings of CATALOG and SCHEMA in the SYSTEM_DEFAULTS table. For more information, see the *SQL/MX Reference Manual*.

Listing the Arguments of the SPJ

Each argument that you list in the CALL statement must correspond with the SQL parameter that you specified in the CREATE PROCEDURE statement.

For example, if you registered the stored procedure with three SQL parameters (two IN parameters and one OUT parameter), you must list three formal parameters, separated by commas, in the CALL statement:

```
CALL samdbcat.persnl.adjustsalary(202, 5, ?);
```

If the SPJ does not accept arguments, you must specify empty parentheses, as shown:

```
CALL samdbcat.sales.lowerprice();
```

Data Conversion of Parameter Arguments

NonStop SQL/MX performs an implicit data conversion when the data type of a parameter argument is compatible with but does not match the formal data type of the stored procedure. For stored procedure input values, the conversion is from the actual argument value to the formal parameter type. For stored procedure output values, the conversion is from the actual output value, which has the data type of the formal parameter, to the declared type of the host variable or dynamic parameter.

Input Parameter Arguments

To pass data to an IN or INOUT parameter of an SPJ, specify an SQL expression that evaluates to a character, date-time, or numeric value. The SQL expression can evaluate to NULL provided that the underlying Java parameter supports null values. For more information, see Null Input and Output (page 52).

For an IN parameter argument, use one of these SQL expressions:

Type of Argument	Examples

Literal	`CALL adjustsalary(202, 5.5, ?);`
	`CALL dailyorders(DATE '2003-03-19', ?);`
	`CALL totalprice(23,'nextday', ?param);`

SQL function (including CASE and CAST expressions)	`CALL dailyorders(CURRENT_DATE, ?);`
Arithmetic expression	`CALL adjustsalary(202,:percent * 0.25, :OUT newsalary);`
Concatenation operation	`CALL totalprice(23,'next' || 'day', ?param);`
Scalar subquery	`CALL totalprice((SELECT qty_ordered FROM odetail WHERE ordernum=100210 AND partnum=5100), 'nextday', ?param);`
Host variable	`CALL adjustsalary(:empnum, :percent, :OUT newsalary);`

Dynamic parameter	`CALL adjustsalary(?, ?, ?);`
	`CALL adjustsalary(?param1, ?param2, ?param3);`

Because an INOUT parameter passes a single value to and accepts a single value from an SPJ, you can specify only host variables or dynamic parameters for INOUT parameter arguments in a CALL statement.

Output Parameter Arguments

An SPJ returns values in OUT and INOUT parameters. Each OUT or INOUT parameter accepts only one value from an SPJ. Any attempt to return more than one value to an output parameter results in a Java exception. See Returning Output Values From the Java Method (page 50).

Specify OUT and INOUT parameter arguments as either host variables in a static CALL statement (for example, `:hostvar`) or dynamic parameters in a dynamic CALL statement (for example, `?` or `?param`).

The parameter arguments that you specify in a CALL statement depend on the application or interface that calls the SPJ. For information about how to invoke SPJs in different applications, see:

- Invoking SPJs in MXCI (page 74)
- Invoking SPJs Statically in an Embedded SQL Program in C, C++, or COBOL (page 76)
- Invoking SPJs Dynamically in an Embedded SQL Program in C, C++, or COBOL (page 77)
- Invoking SPJs in a NonStop ODBC/MX Client (page 80)
- Invoking SPJs in a JDBC/MX Program (page 81)

Invoking SPJs in MXCI

In MXCI, you can invoke an SPJ by issuing a CALL statement directly or by preparing and executing a CALL statement.

Use MXCI named or unnamed parameters anywhere in the argument list of an SPJ invoked in MXCI. An MXCI named parameter is set by the SET PARAM command, and an MXCI unnamed parameter is set by the USING clause of the EXECUTE statement.

You must use an MXCI parameter for an OUT or INOUT parameter argument. MXCI displays all output parameter values after you issue the CALL statement. The procedure call does not change the value of an MXCI named parameter that you use as an OUT or INOUT parameter.

For more information about MXCI parameters, see the *SQL/MX Reference Manual*.

Using MXCI Named Parameters

In an MXCI session, invoke the SPJ named TOTALPRICE, which has two IN parameters and one INOUT parameter. This SPJ accepts the quantity, shipping speed, and price of an item, calculates the total price, including tax and shipping charges, and returns the total price. For more information, see the Sales Class (page 107).

Set the input value for the INOUT parameter by entering a SET PARAM command before calling the SPJ:

```
SET PARAM ?p 10;
CALL samdbcat.sales.totalprice(23, 'standard', ?p);
```

The CALL statement returns the total price of the item:

```
PRICE
---------------------

               253.96

--- SQL operation complete.
```

The value of the MXCI named parameter, `?p`, remains 10.

Using MXCI Unnamed Parameters

In an MXCI session, invoke the SPJ named TOTALPRICE by preparing and executing a CALL statement. The INOUT parameter accepts a value that is set by the USING clause of the EXECUTE statement and returns the total price:

```
SET SCHEMA samdbcat.sales;
PREPARE stmt1 FROM CALL totalprice(50,'nextday',?);
EXECUTE stmt1 USING 2.25;
```

The output of the prepared CALL statement is:

```
PRICE
---------------------

               136.77

--- SQL operation complete.
```

In an MXCI session, invoke the SPJ named TOTALPRICE again by preparing and executing a CALL statement in which all three parameters accept values that are set by the USING clause of the EXECUTE statement. The INOUT parameter returns the total price:

```
PREPARE stmt2 FROM CALL totalprice(?,?,?);
EXECUTE stmt2 USING 3, 'economy', 16.99;
```

The output of the prepared CALL statement is:

```
PRICE
---------------------

                57.12

--- SQL operation complete.
```

Using MXCI with SPJ RS

In an MXCI session, invoking the SPJ named ORDERSUMMARY returns one LARGEINT value as an output parameter and returns a maximum of two result sets. The output parameter row (which in this case contains one column named NUM_ORDERS) is displayed first followed by two stored procedure result sets. Prepare and run the following CALL statement:

```
CALL samdbcat.sales.order_summary('01/01/2001', ?);
```

The output of the prepared call statement is:

```
NUM_ORDERS
--------------------
                  13

ORDERNUM   NUM_PARTS      AMOUNT          Order/Date Last Name
---------- -------------- --------------- ----------  -----------------
-
    100210             4        19020.00 2003-04-10 HUGHES
    100250             4        22625.00 2003-01-23 HUGHES
    101220             4        45525.00 2003-07-21 SCHNABL
    200300             3        52000.00 2003-02-06 SCHAEFFER
    200320             4         9195.00 2003-02-17 KARAJAN
    200490             2         1065.00 2003-03-19 WEIGL
      .
      .
      .
--- 13 row(s) selected.

Order/Num  Part/Num Unit/Price   Qty/Ord    Part Description
---------- -------- ------------ ---------- ------------------
    100210     2001      1100.00          3 GRAPHIC PRINTER,M1
    100210     2403       620.00          6 DAISY PRINTER,T2
    100210      244      3500.00          3 PC GOLD, 30 MB
    100210     5100       150.00         10 MONITOR BW, TYPE 1
    100250     6500        95.00         10 DISK CONTROLLER
    100250     6301       245.00         15 GRAPHIC CARD, HR
      .
      .
      .
--- 70 row(s) selected.
--- SQL operation complete.
```

For more information, see ORDERSUMMARY Stored Procedure (page 121).

NOTE: The SPJ RS feature is supported on systems running J06.05 and later J-series RVUs or H06.16 and later H-series RVUs.

Invoking SPJs Statically in an Embedded SQL Program in C, C++, or COBOL

In an embedded SQL program in C, C++, or COBOL, you can invoke an SPJ in a statically compiled CALL statement. Place a statically compiled CALL statement within an EXEC SQL directive:

```
C/C++
EXEC SQL CALL procedure-name
     ([parameter [{, parameter}...]]);
COBOL
EXEC SQL CALL procedure-name
     ([parameter [{, parameter}...]]) END-EXEC
```

For statically compiled CALL statements, which have static actual parameters, the embedded program does not explicitly refer to input or output descriptors.

In this example of an embedded SQL program in C, the static CALL statement has an IN parameter argument consisting of a host variable, an IN parameter argument of 5.5, and an OUT parameter argument consisting of another host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   NUMERIC(4) hv_empnum_param1;
   double hv_percent_param2;
   NUMERIC(8,2) hv_newsalary_param3;
EXEC SQL END DECLARE SECTION;

/* Set the IN arguments. */
hv_empnum_param1 = 202;
hv_percent_param2 = 5.5;

/* Call the stored procedure.
 * Parameter modes are IN, IN, OUT */
EXEC SQL CALL samdbcat.persnl.adjustsalary(:hv_empnum_param1,
                                            :hv_percent_param2,,
                                           :hv_newsalary_param3);

/* Print the OUT parameter value. */
printf("\nThe new salary is %ld.\n", hv_newsalary_param3);
```

For information about writing embedded SQL programs in C, C++, or COBOL, see the *SQL/MX Programming Manual for C and COBOL*.

Invoking SPJs Dynamically in an Embedded SQL Program in C, C++, or COBOL

In an embedded SQL program in C, C++, or COBOL, you can invoke an SPJ in a dynamically executed CALL statement. For dynamic CALL statements, supply dynamic input data either through an SQL descriptor area or by using host variables. Similarly, you can place dynamic output data either into an SQL descriptor area or directly into host variables. For more information, see:

- Input and Output Descriptors (page 77)
- Argument Lists (page 79)

Input and Output Descriptors

You can use SQL descriptor areas to store and retrieve information about the input and output parameters of a dynamic CALL statement.

Formal and Actual Parameters

A formal parameter is one of the parameters in the formal procedure signature of a CREATE PROCEDURE statement, and an actual parameter is one of the parameter arguments in the procedure call of a CALL statement. More than one actual parameter can map to one formal parameter, so the number of actual parameters might be more than the number of formal parameters.

- Suppose that an SPJ is defined to accept three parameter arguments with PARAMETER_MODE IN, OUT, and IN in this order:

  ```
  CALL myproc(?w /*IN*/, ?x /*OUT*/, ?y + ?z /*IN*/);
  ```

  This CALL statement has three formal parameters (w, x, and y+z) and four actual parameters (w, x, y, and z). The PARAMETER_ORDINAL_POSITION of the parameters is (1, 2, 3, 3).

- Suppose that this CALL statement uses a duplicate actual parameter:

  ```
  CALL myproc(?w /*IN*/, ?x /*OUT*/, ?w + ?y /*IN*/);
  ```

  In this case, there are three actual parameters (w, x, and y). The PARAMETER_ORDINAL_POSITION of the parameters is (1, 2, 3).

Number of Entries in Input and Output Descriptors

In dynamic SQL, the input descriptor has *n* entries, where *n* is the number of actual parameters that map to formal IN and INOUT parameters. *n* is not necessarily the same as the number of formal IN and INOUT parameters.

Conversely, the output descriptor has *m* entries, where *m* is the number of actual parameters that map to formal OUT and INOUT parameters. *m* is the same as the number of formal OUT and INOUT parameters because each output parameter must be represented by a single dynamic parameter in the CALL statement.

Example of Using Input and Output Descriptors

To code an embedded SQL program to use a dynamic CALL statement with SQL descriptor areas:

1. Allocate input and output SQL descriptor areas.

   Allocate an input SQL descriptor area for dynamic input parameters associated with the IN or INOUT mode, and allocate an output descriptor area for dynamic output parameters associated with the OUT or INOUT mode:

   ```
   EXEC SQL ALLOCATE DESCRIPTOR 'in_sda';
   EXEC SQL ALLOCATE DESCRIPTOR 'out_sda';
   ```

   For the syntax of the ALLOCATE DESCRIPTOR statement, see the *SQL/MX Reference Manual*.

2. Prepare the CALL statement.

   Store the CALL statement in a host variable and then prepare the CALL statement. The PREPARE statement checks the statement syntax, determines the data types of parameters, compiles the CALL statement, and associates the CALL statement with a statement name that you can use in a subsequent EXECUTE statement:

   ```
   /* Build the CALL statement in a char buffer. */
   strcpy(hv_sql_stmt,
          "CALL samdbcat.persnl.adjustsalary(?,?,?)");
                                    /* IN, IN, OUT */

   /* Prepare the statement. */
   EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;
   ```

   For the syntax of the PREPARE statement, see the *SQL/MX Reference Manual*.

3. Describe the input and output parameters.

   The DESCRIBE statement stores information about the dynamic input or output parameters of a prepared CALL statement in an SQL descriptor area. Use these DESCRIBE statements to populate the SQL input and output descriptor areas:

   - DESCRIBE INPUT initializes the input SQL descriptor area based on the dynamic input parameters (associated with the IN or INOUT mode) for a prepared CALL statement:

     ```
     EXEC SQL DESCRIBE INPUT sqlstmt
        USING SQL DESCRIPTOR 'in_sda';
     ```

   - DESCRIBE OUTPUT stores descriptions in the output SQL descriptor area of dynamic output parameters (associated with the OUT or INOUT mode) from a prepared CALL statement:

     ```
     EXEC SQL DESCRIBE INPUT sqlstmt
        USING SQL DESCRIPTOR 'in_sda';
     ```

     For the syntax of the DESCRIBE statement, see the *SQL/MX Reference Manual*.

   For the syntax of the DESCRIBE statement, see the *SQL/MX Reference Manual*.

4. Set the input parameter values by using SET DESCRIPTOR.

   Use the SET DESCRIPTOR statement to set information explicitly in the input SQL descriptor area for the individual input parameters:

```
/* Set the input values of the two IN parameters in
 * ordinal positions 1 and 2, which are the only entries
 * in the input descriptor. */
printf("\nEnter the employee number: ";
scanf("%ld", &hv_empnum_param1);
hv_i = 1;
EXEC SQL SET DESCRIPTOR 'in_sda' VALUE :hv_i
   VARIABLE_DATA = :hv_empnum_param1;

printf("\nEnter the percentage adjustment: ";
scanf("%f", &hv_percent_param2);
hv_i = 2;
EXEC SQL SET DESCRIPTOR 'in_sda' VALUE :hv_i
   VARIABLE_DATA = :hv_percent_param2;
```

For the syntax of the SET DESCRIPTOR statement, see the *SQL/MX Reference Manual*.

NOTE: You cannot modify the PARAMETER_ORDINAL_POSITION and PARAMETER_MODE descriptor items.

5. Execute the prepared CALL statement.

   Issue an EXECUTE statement that has an input USING clause for the input SQL descriptor area and an output INTO clause for the output SQL descriptor area:

   ```
   EXEC SQL EXECUTE sqlstmt
      USING SQL DESCRIPTOR 'in_sda'
      INTO SQL DESCRIPTOR 'out_sda';
   ```

   For the syntax of the EXECUTE statement, see the *SQL/MX Reference Manual*.

6. Retrieve the output parameter value by using GET DESCRIPTOR.

   Use the GET DESCRIPTOR statement to retrieve information for the output parameter from the output SQL descriptor area:

   ```
   hv_i = 1;
   EXEC SQL GET DESCRIPTOR 'out_sda' VALUE :hv_i
      VARIABLE_DATA = :hv_newsalary_param3;
   printf("\nThe new salary is %ld.\n", hv_newsalary_param3);
   ```

   For the syntax of the GET DESCRIPTOR statement, see the *SQL/MX Reference Manual*.

7. Retrieve parameter descriptor information by using GET DESCRIPTOR.

   Aside from obtaining data about the output dynamic parameters, you can also use the GET DESCRIPTOR statement to retrieve the PARAMETER_MODE and PARAMETER_ORDINAL_POSITION descriptor items for dynamic parameters.

   In this example, the GET DESCRIPTOR statement stores the PARAMETER_MODE and PARAMETER_ORDINAL_POSITION information in designated host variables for the OUT parameter:

   ```
   hv_i = 1;
   EXEC SQL GET DESCRIPTOR 'out_sda' VALUE :hv_i
      VARIABLE_DATA = :hv_newsalary_param3
      PARAMETER_MODE = :pm_newsalary_param3
      PARAMETER_ORDINAL_POSITION = :pop_newsalary_param3;
   ```

   For more information about descriptor items, see the *SQL/MX Reference Manual*.

For more information about writing embedded SQL programs in C, C++, or COBOL, see the *SQL/MX Programming Manual for C and COBOL*.

Argument Lists

You can execute an embedded dynamic CALL statement without input or output descriptors. Instead, pass two argument lists in the input USING clause and the output INTO clause directly to the EXECUTE statement.

In this example, an embedded dynamic CALL statement uses argument lists:

```
EXEC SQL BEGIN DECLARE SECTION;
   NUMERIC(4) hv_empnum_param1;
   double hv_percent_param2;
   NUMERIC(8,2) hv_newsalary_param3;
   CHAR hv_sql_stmt[64];
EXEC SQL END DECLARE SECTION;

/* Build the CALL statement in a char buffer. */
strcpy(hv_sql_stmt, "CALL samdbcat.persnl.adjustsalary(?,?,?)");
                                                /* IN, IN, OUT */

/* Prepare the statement. */
EXEC SQL PREPARE sqlstmt FROM :hv_sql_stmt;

/* Set the input values of the two IN parameters in
 * ordinal positions 1 and 2.
printf("\nEnter the employee number: ";
scanf("%ld", &hv_empnum_param1);
printf("\nEnter the percentage adjustment: ";
scanf("%f", &hv_percent_param2);

/* Call the stored procedure. */
EXEC SQL EXECUTE sqlstmt
   USING :hv_empnum_param1, :hv_percent_param2  /* IN, IN */
   INTO  :hv_newsalary_param3;                  /* OUT */

printf("\nThe new salary is %ld.\n", hv_newsalary_param3);
```

For more information about writing embedded SQL programs in C, C++, or COBOL, see the *SQL/MX Programming Manual for C and COBOL*.

Invoking SPJs in a NonStop ODBC/MX Client

You can execute a CALL statement in a NonStop ODBC/MX client. Microsoft ODBC requires that you put the CALL statement in an escape clause:

```
{call procedure-name ([parameter][,parameter]]...)}
```

For IN or INOUT parameters, use a literal or a parameter marker (?).

NOTE: You cannot use an empty string as an IN or INOUT parameter in the argument list.

If you specify a literal for an INOUT parameter, the driver discards the output value.

For OUT parameters, you can use only a parameter marker (?). You must bind all parameter markers with the `SQLBindParameter` function before you can execute the CALL statement.

In this example, a CALL statement is executed from a NonStop ODBC/MX client:

```
/* Declare variables. */
SQLHSTMT hstmt;
SQL_NUMERIC_STRUCT salary;
SDWORD cbParam = SQL_NTS;

/* Bind the parameter markers. */
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_NUMERIC, SQL_NUMERIC,
4, 0, 202, 0, &cbParam);

SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_FLOAT,
0, 0, 5.5, 0, &cbParam);

SQLBindParameter(hstmt, 3, SQL_PARAM_OUTPUT, SQL_C_NUMERIC, SQL_NUMERIC,
8, 2, &salary, 0, &cbParam);
```

```
/* Execute the CALL statement. */
SQLExecDirect(hstmt, "{call adjustsalary(?,?,?)}", SQL_NTS);
```

NOTE: Nonstop ODBC/MX Client does not support SQLExecDirect API for Stored Procedure in Java with Result Sets (SPJ RS).

For more information about NonStop ODBC/MX clients, see the *ODBC/MX Driver for Windows Manual*.

Invoking SPJs in a JDBC/MX Program

You can execute a CALL statement in a JDBC/MX program by using the JDBC `CallableStatement` interface. JDBC/MX requires that you put the CALL statement in an escape clause:

```
{call procedure-name([parameter[{, parameter}...]])}
```

Set input values for IN and INOUT parameters by using the `set`*`type`*`()` methods of the `CallableStatement` interface.

Retrieve output values from OUT and INOUT parameters by using the `get`*`type`*`()` methods of the `CallableStatement` interface.

If the parameter mode is OUT or INOUT, you must register the parameter as an output parameter by using the `registerOutParameter()` method of the `CallableStatement` interface before executing the CALL statement.

In this example, a CALL statement is executed from a JDBC/MX program:

```
CallableStatement stmt =
   con.prepareCall("{call adjustsalary(?,?,?)}");

stmt.setBigDecimal(1,202); // x = 202
stmt.setDouble(2,5.5); // y = 5.5
stmt.registerOutParameter(3, java.sql.Types.NUMERIC);

stmt.execute();

int z = stmt.getBigDecimal(3); // Retrieve the value of the
                               // OUT parameter
```

For more information about JDBC/MX and mappings of SQL/MX to JDBC d ata types, see the *JDBC Driver for SQL/MX Programmer's Reference*.

Invoking SPJs in a Trigger

A trigger is a mechanism in the database that enables the database engine to perform certain actions when a specified event occurs. SPJs are useful as triggered actions, because they can help you encapsulate and enforce rules in the database. For more information about the benefits of using SPJs, see Benefits of SPJs (page 19). Nonstop SQL/MX supports a CALL statement in a trigger, provided that the SPJ in the CALL statement does not have any OUT or INOUT parameters or return any result sets. For information about OUT and INOUT parameters, see Returning Output Values From the Java Method (page 50) and Output Parameter Arguments (page 74). For more information about result sets, see Stored Procedure Result Sets (page 51).

Example

Consider a library environment, where each time a member borrows a book, the member is charged a fee. If the fees exceeds the credit limit, the student is blocked. The Database Administrator (DBA) needs to update the dues and block the students who have crossed the limit of credit. Also, DBA needs to be informed if a user has been blocked by some external means. One might think that a SIGNAL statement can be used for this task to emit a message to the console for the same. The problem with using SIGNAL statement in this case is that the encompassing transaction is aborted, and therefore the dues will not be updated properly. DBA wants to update the dues but at the same time, flag the member as being blocked. One solution is to use a CALL statement and delegate the implementation of the external notification to a member-defined routine.

Consider a stored procedure called INFORM_DBA_FUNC, that writes a message to a file or sends an e-mail to the DBA with the list of members who have been blocked (member names are sent as an argument to the routine).

```
CREATE TRIGGER INFORM_DBA AFTER UPDATE OF (ISBLOCKED)
ON MEMBER
REFERENCING NEW AS NEWR
FOR EACH ROW
WHEN (NEWR.BLOCK <> 0)
CALL INFORM_DBA_FUNC (NEWR.MEMBERNAME);
```

For information about the CREATE TRIGGER syntax, see the *HP Nonstop SQL/MX Release 3.2 Reference Manual.*

6 Managing SPJs in NonStop SQL/MX

This section covers management tasks related to SPJs in NonStop SQL/MX:

Granting Privileges for Invoking SPJs

Security for SPJs is implemented by the schema ownership rules and by granting privileges to specified users.

The schema in which an SPJ is registered is the unit of ownership. The person who creates the schema is the owner of that schema and all objects associated with it. In NonStop SQL/MX, the schema owner and the super ID automatically have these privileges:

- Ability to create and drop SPJs in the schema
- EXECUTE and WITH GRANT OPTION privileges on the SPJs in the schema

To create or drop an SPJ, you must be either the owner of its schema or the super ID. To invoke an SPJ, you must have the EXECUTE privilege on the SPJ. The EXECUTE privilege allows a user to invoke an SPJ by issuing a CALL statement. The WITH GRANT OPTION privilege allows a user to grant the EXECUTE and WITH GRANT OPTION privileges to other users. For more information, see:

- Granting Privileges on an SPJ (page 83)
- Granting Privileges on Referenced Database Objects (page 84)
- Revoking Privileges on an SPJ (page 85)

To display the current ownership and privileges, see Showing Privileges on the SPJs (page 89).

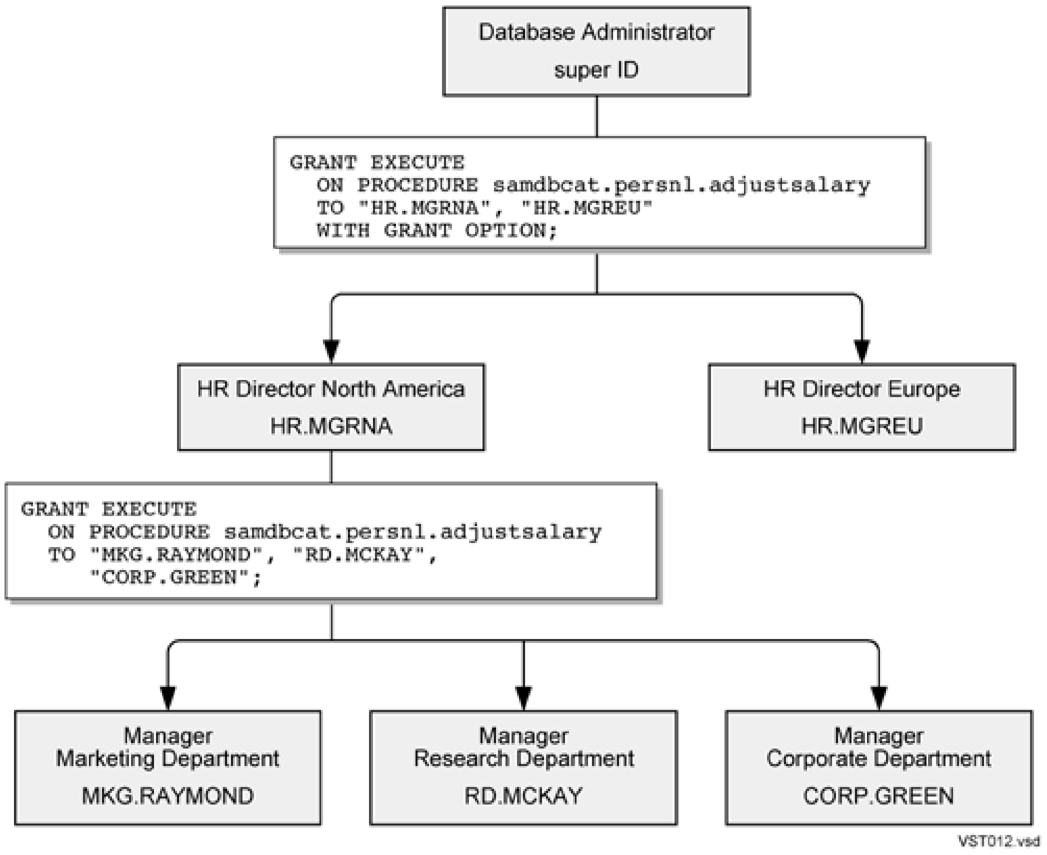
Granting Privileges on an SPJ

Use the GRANT EXECUTE or GRANT statement to assign the EXECUTE and WITH GRANT OPTION privileges on an SPJ to specific users. In a GRANT statement, specify ALL PRIVILEGES to grant the EXECUTE privilege on an SPJ. For the syntax of the GRANT EXECUTE and GRANT statements, see the *SQL/MX Reference Manual*.

If you own the SPJ, or are the super ID acting on behalf of the object owner, you can grant the EXECUTE and WITH GRANT OPTION privileges on the SPJ to any user. If you are not the owner of the SPJ or the super ID, you must have been granted the WITH GRANT OPTION privilege to grant privileges to other users.

As the owner of an SPJ, or the super ID acting on behalf of a user with the WITH GRANT OPTION privilege, you can selectively grant the EXECUTE and WITH GRANT OPTION privileges to specified users. For some SPJs, particularly ones that handle sensitive information or modify data, you should grant the EXECUTE and WITH GRANT OPTION privileges to a restricted number of users.

For example, the SPJ named ADJUSTSALARY changes an employee's salary in the database. Therefore, only specific users should be allowed to invoke this SPJ. In this example, the database administrator (the super ID), acting on behalf of the SPJ owner, grants the EXECUTE and WITH GRANT OPTION privileges on ADJUSTSALARY to the regional HR directors. The HR directors grant the EXECUTE privilege on ADJUSTSALARY to the department managers:



In some cases, all users of a database system might need to invoke an SPJ. For example, the SPJ named MONTHLYORDERS determines the number of product orders during a given month. This SPJ does not handle sensitive information or modify data and might be useful to anyone within the company. Therefore, the database administrator (the super ID), acting on behalf of the SPJ owner, grants the EXECUTE privilege on MONTHLYORDERS to PUBLIC, meaning all present and future user IDs:

```
GRANT EXECUTE
  ON samdbcat.sales.monthlyorders
  TO PUBLIC;
```

After granting the EXECUTE privilege to PUBLIC, you cannot revoke the privilege from a subset of users. You must revoke the privilege from PUBLIC and then grant the privilege to specific users, excluding users who should not have the privilege.

Granting Privileges on Referenced Database Objects

If the SPJ operates on a database object, the user ID that invokes the SPJ must have the appropriate privileges on that database object.

For example, users with the EXECUTE privilege on the SPJ named ADJUSTSALARY, which selects data from and updates a table, must have the SELECT and UPDATE privileges on the SQL/MX table named EMPLOYEE. The database administrator (the super ID), acting on behalf of the object owner, grants these access privileges to regional HR directors:

```
GRANT SELECT, UPDATE (salary)
  ON TABLE samdbcat.persnl.employee
  TO "HR.MGRNA", "HR.MGREU"
  WITH GRANT OPTION;
```

The HR director of North America then grants these access privileges to department managers:

```
GRANT SELECT, UPDATE (salary)
  ON TABLE samdbcat.persnl.employee
  TO "MKG.RAYMOND", "RD.MCKAY", "CORP.GREEN";
```

All users with the EXECUTE privilege on the SPJ named MONTHLYORDERS must have the SELECT privilege on the ORDERS table:

```
GRANT SELECT
  ON TABLE samdbcat.sales.orders
  TO PUBLIC;
```

The types of SQL statements in the underlying SPJ method, such as SELECT, UPDATE, DELETE, and INSERT, should indicate which privileges, such as SELECT, UPDATE, DELETE, and INSERT, are required for the referenced database objects.

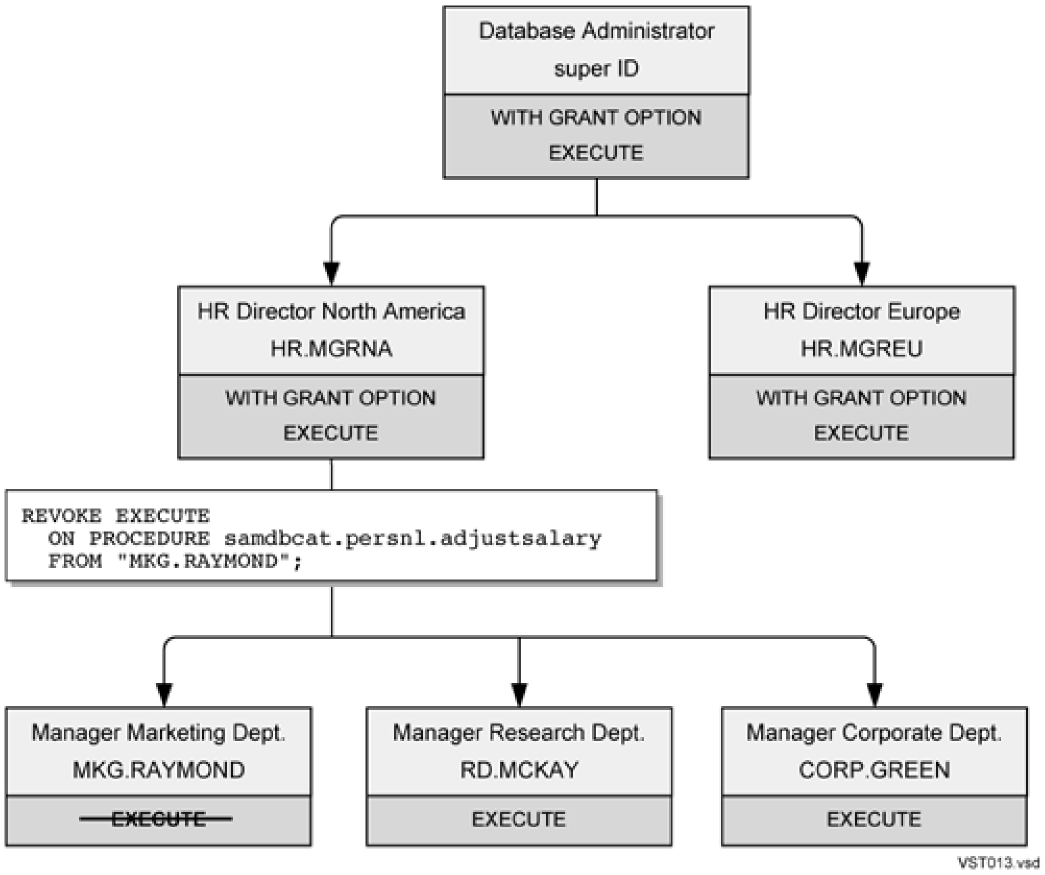
For the syntax of the GRANT statement, see the *SQL/MX Reference Manual*. For more information about granting privileges on SQL/MX database objects, see the SQL/MX Installation and Upgrade Guide and SQL/MX Management Manual. For information about securing SQL/MP database objects, see the *SQL/MP Installation and Management Guide*.

Revoking Privileges on an SPJ

Use the REVOKE EXECUTE or REVOKE statement to remove the EXECUTE or WITH GRANT OPTION privilege on an SPJ from specific users. In a REVOKE statement, specify ALL PRIVILEGES to revoke the EXECUTE privilege on an SPJ. For the syntax of the REVOKE EXECUTE and REVOKE statements, see the *SQL/MX Reference Manual*.

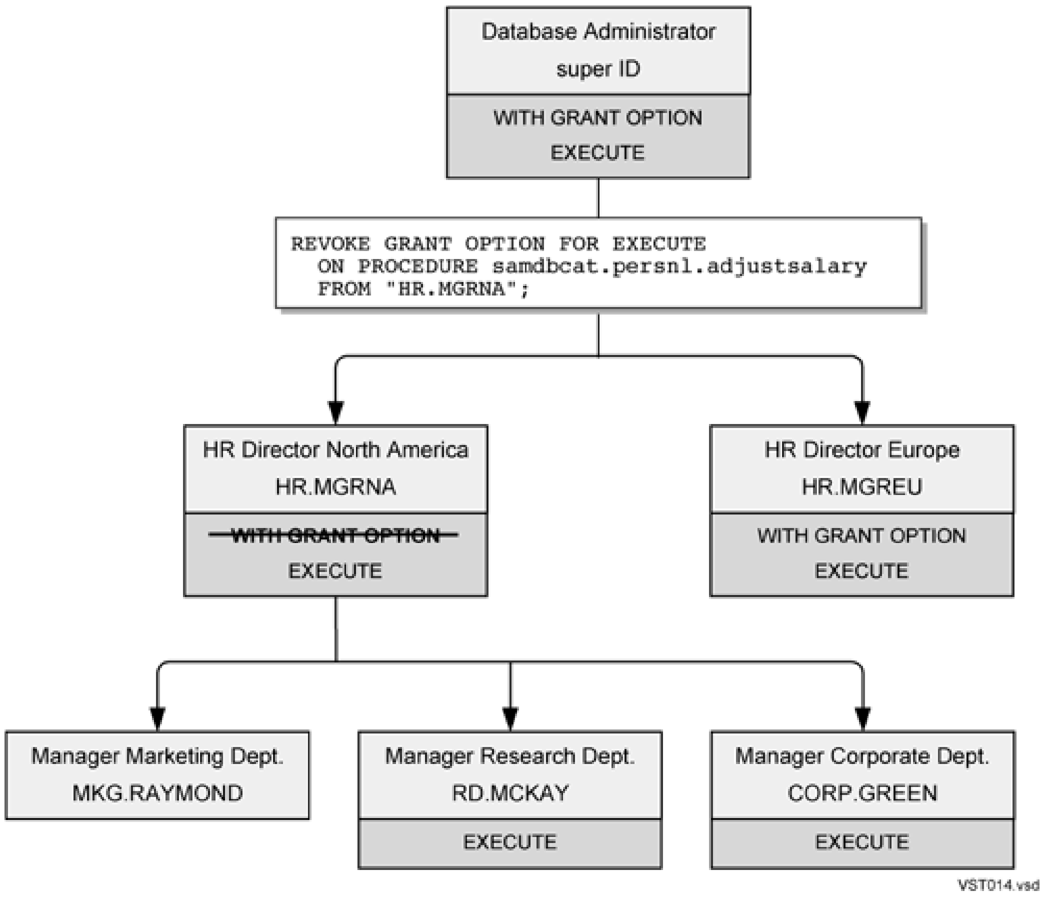
If you own the SPJ, you can revoke the EXECUTE and WITH GRANT OPTION privileges on the SPJ from any user to whom you granted those privileges. If you are the super ID acting on behalf of a grantor, you can revoke the EXECUTE and WITH GRANT OPTION privileges on the SPJ from any user to whom the grantor granted those privileges. If you are not the owner of the SPJ or the super ID, you must have been granted the WITH GRANT OPTION privilege to revoke privileges from other users, and you can revoke privileges only from other users to whom you have granted privileges.

For example, the HR director of North America can revoke the EXECUTE privilege on ADJUSTSALARY from one or more of the department managers to whom the director granted privileges. In this example, the HR director revokes the EXECUTE privilege from the Marketing Department manager:

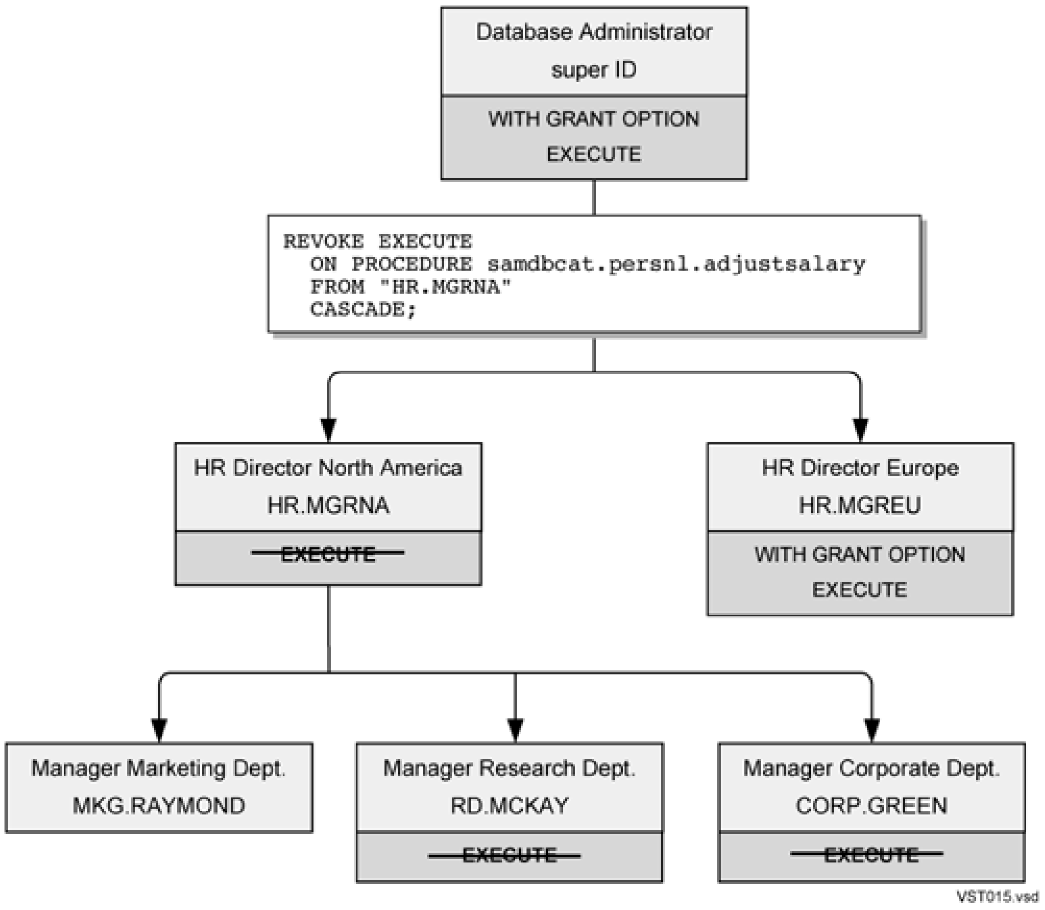


The HR director of North America cannot revoke the EXECUTE or WITH GRANT OPTION privilege from the HR director of Europe because it was the database administrator who granted the privileges on behalf of the SPJ owner.

The database administrator (the super ID), acting on behalf of the grantor of the privilege, can revoke the WITH GRANT OPTION privilege on ADJUSTSALARY from any user with this privilege. In this example, the database administrator, acting on behalf of the SPJ owner, revokes the WITH GRANT OPTION privilege from the HR director of North America:



The database administrator, acting on behalf of the grantor of the privilege, can also revoke the EXECUTE privilege from any user with this privilege and from any dependent privileges by using the CASCADE option. In this example, the database administrator, acting on behalf of the SPJ owner, revokes the EXECUTE privilege from the HR director of North America and from the department managers to whom the HR director granted privileges:



For SPJs on which all users (that is, PUBLIC) have privileges, you can revoke privileges from PUBLIC but not from one or more specific users. For example, this statement revokes the EXECUTE privilege on the SPJ named MONTHLYORDERS from all users (that is, PUBLIC):

```
REVOKE EXECUTE
  ON PROCEDURE samdbcat.sales.monthlyorders
  FROM PUBLIC;
```

Displaying Information About SPJs

To manage SPJs in NonStop SQL/MX, you might need to gather information about the SPJs that are already registered in the database. For more information, see:

- Listing the SPJs in a Catalog (page 88)
- Showing Privileges on the SPJs (page 89)
- Showing the Procedure Labels of All SPJs in a Catalog (page 92)
- Showing the Syntax of an SPJ (page 92)

The first three topics suggest querying SQL/MX system metadata. For information about system metadata tables, see the *SQL/MX Reference Manual*.

Listing the SPJs in a Catalog

To display the names of all SPJs in a user catalog of the database, query the OBJECTS, SCHEMATA, and CATSYS system metadata tables and store the results in a log file, if desired.

For example:

1. In an MXCI session, enter the LOG command so that MXCI writes the output of the query to a log file:

```
LOG spj_list.txt;
```

2. In the same MXCI session, enter this query to list all the SPJs in a catalog of the database:

```
SELECT SUBSTRING(cat_name,1,25) AS "CATALOG",
       SUBSTRING(schema_name,1,25) AS "SCHEMA",
       SUBSTRING(object_name,1,25) AS "PROCEDURE"
  FROM catalog.definition_schema_version_vernum.objects ot,
       nonstop_sqlmx_node.system_schema.schemat ast,
       nonstop_sqlmx_node.system_schema.catsys ct
  WHERE ot.object_type = 'UR'
    AND ot.schema_uid = st.schema_uid
    AND st.cat_uid = ct.cat_uid;
```

   The schema version number *vernum* for NonStop SQL/MX Releases 2.0, 2.1, and 2.2 is 1200.

3. To end logging, enter the LOG command without the name of the log file:

```
LOG;
```

The query returns the catalog, schema, and procedure names of all the SPJs in the catalog, as shown:

```
CATALOG                   SCHEMA                    PROCEDURE
------------------------- ------------------------- --------------------

SAMDBCAT                  PERSNL                    ADJUSTSALARY
SAMDBCAT                  SALES                     DAILYORDERS
SAMDBCAT                  SALES                     LOWERPRICE
SAMDBCAT                  SALES                     MONTHLYORDERS
SAMDBCAT                  SALES                     TOTALPRICE

--- 5 row(s) selected.
```

Showing Privileges on the SPJs

To display the ownership and privileges for all the SPJs in a user catalog of the database, query the TBL_PRIVILEGES, OBJECTS, and SCHEMATA system metadata tables and store the results in a log file, if desired.

For example:

1. In an MXCI session, enter the LOG command so that MXCI writes the output of the query to a log file:

```
LOG spj_prvlgs.txt;
```

2. In the same MXCI session, enter this query to return ownership and grantor information for the SPJs in a catalog of the database:

```
SELECT SUBSTRING(schema_name,1,8) AS "SCHEMA",
       SUBSTRING(object_name,1,15) AS "PROCEDURE",
       object_owner, grantor, grantor_type, grantee
  FROM catalog.definition_schema_version_vernum.objects ot,
       nonstop_sqlmx_node.system_schema.schemata st,
     catalog.definition_schema_version_vernum.tbl_privileges pr
  WHERE ot.object_type = 'UR'
    AND ot.schema_uid = st.schema_uid
    AND ot.object_uid = pr.table_uid;
```

   The schema version number *vernum* for NonStop SQL/MX Releases 2.0, 2.1, and 2.2 is 1200.

   For the output, see Interpreting Ownership and Grantor Informationntor Information (page 90).

3. Enter this query to return the privileges that grantees have on the SPJs in a catalog of the database:

```
SELECT SUBSTRING(schema_name,1,8) AS "SCHEMA",
       SUBSTRING(object_name,1,15) AS "PROCEDURE",
       grantee, grantee_type, privilege_type, is_grantable
  FROM catalog.definition_schema_version_vernum.objects ot,
       nonstop_sqlmx_node.system_schema.schematast,
   catalog.definition_schema_version_vernum.tbl_privileges pr
  WHERE ot.object_type = 'UR'
    AND ot.schema_uid = st.schema_uid
    AND ot.object_uid = pr.table_uid;
```

The schema version number *vernum* for NonStop SQL/MX Releases 2.0, 2.1, and 2.2 is 1200.

For the output, see Interpreting Privileges Information (page 90).

4. To end logging, enter the LOG command without the name of the log file:

```
LOG;
```

Interpreting Ownership and Grantor Informationntor Information

The first query in 2 returns the owners, grantors of privileges, and grantees for all SPJs in a catalog, as shown:

```
SCHEMA    PROCEDURE        OBJECT_OWNER  GRANTOR      GRANTOR_TYPE GRANTEE
--------  ---------------  ------------  -----------  ------------ ------
PERSNL    ADJUSTSALARY             2053           -2  S              2053
PERSNL    ADJUSTSALARY             2053         2054  U              2069
PERSNL    ADJUSTSALARY             2053         2054  U              2068
PERSNL    ADJUSTSALARY             2053         2054  U              2055
PERSNL    ADJUSTSALARY             2053         2053  U              2054
PERSNL    ADJUSTSALARY             2053         2053  U              2051
SALES     DAILYORDERS              2053           -2  S              2053
SALES     LOWERPRICE               2053           -2  S              2053
SALES     MONTHLYORDERS            2053           -2  S              2053
SALES     MONTHLYORDERS            2053         2053  U                -1
SALES     TOTALPRICE               2053           -2  S              2053

--- 11 row(s) selected.
```

The owners, grantors, and grantees are represented by user ID numbers. Use the USER_GETINFO_ procedure to return the user name associated with the user ID number. For more information, see the *Guardian Procedure Calls Reference Manual*.

By default, the system (represented by -2 in the GRANTOR column) grants ownership privileges to the creator of each schema and its SPJs, who happens to be user ID 2053 in this case, as shown in the OBJECT_OWNER column. The GRANTOR_TYPE column shows whether the system (S) or a user (U) was responsible for granting privileges to other users.

In the example, the super ID (65535), acting on behalf of the object owner (2053), granted privileges on the MONTHLYORDERS procedure to PUBLIC (represented by -1 in the GRANTEE column). The super ID, acting on behalf of the object owner (2053), also granted privileges on the ADJUSTSALARY procedure to HR.MGRNA (2054) and HR.MGREU (2051). The user named HR.MGRNA (2054) granted privileges to MKG.RAYMOND (2069), RD.MCKAY (2068), and CORP.GREEN (2055), as shown in the highlighted rows.

Interpreting Privileges Information

The second query in 3 returns the grantees and the privileges they have on each SPJ in a catalog, as shown:

```
SCHEMA    PROCEDURE       GRANTEE   GRANTEE_TYPE  PRIVILEGE_TYPE  IS_GRANTABLE
--------  --------------  --------  ------------  --------------  ------------
```

```
PERSNL    ADJUSTSALARY        2053  U             E               Y
PERSNL    ADJUSTSALARY        2069  U             E               N
PERSNL    ADJUSTSALARY        2068  U             E               N
PERSNL    ADJUSTSALARY        2055  U             E               N
PERSNL    ADJUSTSALARY        2054  U             E               Y
PERSNL    ADJUSTSALARY        2051  U             E               Y
SALES     DAILYORDERS         2053  U             E               Y
SALES     LOWERPRICE          2053  U             E               Y
SALES     MONTHLYORDERS       2053  U             E               Y
SALES     MONTHLYORDERS         -1  P             E               N
SALES     TOTALPRICE          2053  U             E               Y

--- 11 row(s) selected.
```

The grantees are represented by user ID numbers. Use the USER_GETINFO_ procedure to return the user name associated with the user ID number. For more information, see the *Guardian Procedure Calls Reference Manual*.

By default, the owner of each SPJ (which is 2053 in this case) has these privileges:

- EXECUTE privilege, represented by E in the PRIVILEGE_TYPE column
- WITH GRANT OPTION privilege, represented by Y in the IS_GRANTABLE column

In the example, these users have the EXECUTE (E) and WITH GRANT OPTION (Y) privileges on the ADJUSTSALARY procedure:

- SPJ owner (2053)
- The super ID (65535)
- HR.MGRNA (2054)
- HR.MGREU (2051)

These users have the EXECUTE (E) privilege but not the WITH GRANT OPTION (N) privilege on the ADJUSTSALARY procedure:

- MKG.RAYMOND (2069)
- RD.MCKAY (2068)
- CORP.GREEN (2055)

As shown in the highlighted row, all users of the system (that is, PUBLIC users) have the EXECUTE (E) privilege on the MONTHLYORDERS procedure. The GRANTEE_TYPE column shows whether the grantee is a public grant (P) or a user grant (U).

Showing the Procedure Label

The procedure label is used internally by NonStop SQL/MX to track privileges on an SPJ. NonStop SQL/MX creates one procedure label for each SPJ. The procedure label must exist and be available for an SPJ to be invoked successfully. Knowing the locations of the procedure labels is particularly important when you are using SPJs in a distributed database environment. For more information, see Using SPJs in a Distributed Database Environment (page 98).

Showing the Procedure Label of One SPJ

To show the location of the procedure label for a specific SPJ, use the SHOWDDL command. For example, enter this command in MXCI:

```
SHOWDDL PROCEDURE samdbcat.sales.lowerprice;
```

The SHOWDDL command returns information about the procedure label in the LOCATION clause:

```
CREATE PROCEDURE SAMDBCAT.SALES.LOWERPRICE
  (
  )
  EXTERNAL NAME 'Sales.lowerPrice ()'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LOCATION \KINGPIN.$TX0115.ZSDX7KT4.SL9FSB00
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA
  NOT DETERMINISTIC
  ISOLATE
  ;

--- SQL operation complete.
```

For more information about the SHOWDDL command, see Showing the Syntax of an SPJ (page 92).

Showing the Procedure Labels of All SPJs in a Catalog

To show the locations of procedure labels for all SPJs in a user catalog of the database, query the OBJECTS, SCHEMATA, and REPLICAS metadata tables:

```
SELECT SUBSTRING(schema_name,1,8) AS "SCHEMA",
       SUBSTRING(object_name,1,15) AS "PROCEDURE",
       system_name, data_source, file_suffix
  FROM catalog.definition_schema_version_vernum.objects ot,
       nonstop_sqlmx_node.system_schema.schemata st,
       catalog.definition_schema_version_vernum.replicas rt
  WHERE ot.object_type = 'UR'
    AND ot.schema_uid = st.schema_uid
    AND ot.object_uid = rt.object_uid;
```

The schema version number `vernum` for NonStop SQL/MX Releases 2.0, 2.1, and 2.2 is 1200.

The query returns the names of the schema, procedure, node (SYSTEM_NAME), volume (DATA_SOURCE), and the subvolume and file (FILE_SUFFIX), as shown:

```

SCHEMA    PROCEDURE        SYSTEM_NAME  DATA_SOURCE  FILE_SUFFIX
--------  ---------------  -----------  -----------  ------------------

PERSNL    ADJUSTSALARY     \KINGPIN     $TX0115      ZSDPK4GV.Q85DXB00
SALES     DAILYORDERS      \KINGPIN     $TX0115      ZSDX7KT4.N71HTB00
SALES     LOWERPRICE       \KINGPIN     $TX0115      ZSDX7KT4.SL9FSB00
SALES     MONTHLYORDERS    \KINGPIN     $TX0115      ZSDX7KT4.G13HVB00
SALES     TOTALPRICE       \KINGPIN     $TX0115      ZSDX7KT4.T4BGWB00

--- 5 row(s) selected.
```

Showing the Syntax of an SPJ

To show the CREATE PROCEDURE syntax of an SPJ as it exists in system metadata, use the SHOWDDL command. The SHOWDDL command generates a description of the metadata for an SPJ and displays it in an MXCI session.

Use the SHOWDDL command to display information about SPJs that you have migrated to the system from an earlier release of NonStop SQL/MX. For more information, see the *SQL/MX Database and Application Migration Guide*.

You can also use the SHOWDDL command to create a replica of an SPJ. For more information, see Regenerating CREATE PROCEDURE Statements (page 96).

Using the SHOWDDL Command

To display information about the SPJ named ADJUSTSALARY, for example, enter this SHOWDDL command in an MXCI session:

```
SHOWDDL PROCEDURE samdbcat.persnl.adjustsalary;
```

For the syntax of the SHOWDDL command, see the *SQL/MX Reference Manual*.

Output of the SHOWDDL Command

The SHOWDDL command displays information about an SPJ in this format:

```
CREATE PROCEDURE catalog-name.
schema-name.procedure-name
  (
    [sql-parameter
[{, sql-parameter}...]]
  )
  EXTERNAL NAME 'java-method-name (
java-signature)'
  EXTERNAL PATH 'class-or-JAR-file-path'
  LOCATION procedure-label
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  {NO SQL | MODIFIES SQL DATA | READS SQL DATA
    | CONTAINS SQL}
  {DETERMINISTIC | NOT DETERMINISTIC}
  {NO ISOLATE | ISOLATE}
  ;
```

The output of the SHOWDDL command does not exactly match the original CREATE PROCEDURE statement in these ways:

- The SHOWDDL command does not omit the optional clauses, such as LOCATION, CONTAINS SQL, NOT DETERMINISTIC, and ISOLATE.
- The SHOWDDL command always generates a Java signature for the SPJ.
- The SHOWDDL command displays DECIMAL or NUMERIC data types instead of PIC S9, when PIC S9 is specified for an SQL parameter, because NonStop SQL/MX stores PIC S9 as one of these data types.
- The SHOWDDL command displays CHARACTER data types instead of PIC X, when PIC X is specified for an SQL parameter, because NonStop SQL/MX stores PIC X as a CHAR data type.
- The SHOWDDL command displays the NCHAR data type as a CHAR type with UCS2 (or ISO88591, KANJI, or KSC5601) as the CHARACTER SET modifier.
- All ANSI names are fully qualified.
- All physical location names are fully expanded.

The SHOWDDL command for an SPJ named ADJUSTSALARY returns this output:

```
CREATE PROCEDURE SAMDBCAT.PERSNL.ADJUSTSALARY
  (
    IN EMPNUM NUMERIC(4)
  , IN PERCENT DOUBLE PRECISION
  , OUT NEWSALARY NUMERIC(8,2)
  )
  EXTERNAL NAME 'Payroll.adjustSalary
(java.math.BigDecimal,double,java.math.BigDecimal[])'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LOCATION \KINGPIN.$TX0115.ZSDPK4GV.Q85DXB00
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA
  NOT DETERMINISTIC
  ISOLATE
  ;

--- SQL operation complete.
```

For more information about ADJUSTSALARY and its underlying SPJ method, `adjustSalary()`, see Appendix A: Sample SPJs.

Output of the SHOWDDL Command of an SPJ with Result Sets

To display information about an SPJ that returns result sets (in this example, ORDER_SUMMARY), enter this SHOWDDL command in an MXCI session:

```
SHOWDDL PROCEDURE samdbcat.sales.order_summary;
```

The SHOWDDL command for the ORDER_SUMMARY SPJ returns this output:

```
CREATE PROCEDURE SAMDBCAT.SALES.ORDER_SUMMARY
(
IN ON_OR_AFTER_DATE VARCHAR(20) CHARACTER SET ISO88591
, OUT NUM_ORDERS LARGEINT
)
EXTERNAL NAME 'SPJMethods.orderSummary
(java.lang.String,long[],java.sql.ResultSet[],java.sql.ResultSet[])'
EXTERNAL PATH '/usr/mydir/myclasses'
LOCATION \ALPINE.$SYSTEM.ZSDCR2C6.L1Z7NW00
LANGUAGE JAVA
PARAMETER STYLE JAVA
READS SQL DATA
DYNAMIC RESULT SETS 2
NOT DETERMINISTIC
ISOLATE
;
--- SQL operation complete.
```

For more information about ORDER_SUMMARY and its underlying SPJ method, orderSummary(), see Appendix A: Sample SPJs.

Keeping SPJ Statements in OBEY Command Files

Consider keeping your SPJ statements in OBEY command files. That way, you can quickly and easily create, drop, or reregister SPJs and grant or revoke privileges to the SPJs, as needed. For more information, see:

- OBEY Command Files (page 94)
- Regenerating CREATE PROCEDURE Statements (page 96)

OBEY Command Files

An OBEY command file is either an OSS text file (odd-unstructured file, type 180) or a Guardian EDIT file (type 101). For more information about the OBEY command, see the *SQL/MX Reference Manual*.

CREATE PROCEDURE Statements in an OBEY Command File

For example, the OBEY command file, `createprocs.sql`, contains a series of CREATE PROCEDURE statements:

```
?SECTION "CREATE SALES SPJs"

  CREATE PROCEDURE samdbcat.sales.lowerprice()
    EXTERNAL NAME 'Sales.lowerPrice'
    EXTERNAL PATH '/usr/mydir/myclasses'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    MODIFIES SQL DATA;
...


?SECTION "CREATE PERSNL SPJs"
```

```
  CREATE PROCEDURE samdbcat.persnl.adjustsalary(IN empnum NUMERIC(4),
                                                IN percent FLOAT,
                                               OUT newsalary NUMERIC(8,2))
    EXTERNAL NAME 'Payroll.adjustSalary'
    EXTERNAL PATH '/usr/mydir/myclasses'
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    MODIFIES SQL DATA;
...
```

To create SPJs from an OBEY command file, issue the OBEY command in MXCI:

```
OBEY createprocs.sql;
```

NOTE: The SPJ class must exist on the disk for the CREATE PROCEDURE statements to run successfully.

DROP PROCEDURE Statements in an OBEY Command File

You can use another or the same OBEY command file to drop a series of SPJs. For example, the OBEY command file, `dropprocs.sql`, contains a series of DROP PROCEDURE statements:

```
?SECTION "DROP SALES SPJs"

  DROP PROCEDURE samdbcat.sales.lowerprice;
  DROP PROCEDURE samdbcat.sales.dailyorders;
  DROP PROCEDURE samdbcat.sales.monthlyorders;
  DROP PROCEDURE samdbcat.sales.totalprice;


?SECTION "DROP PERSNL SPJs"

  DROP PROCEDURE samdbcat.persnl.adjustsalary;
```

To drop SPJs, issue the OBEY command in MXCI:

```
OBEY dropprocs.sql "DROP SALES SPJs";
```

GRANT EXECUTE Statements in an OBEY Command File

You can use another or the same OBEY command file to grant privileges on a series of SPJs. For example, the OBEY command file, `grantprocs.sql`, contains a series of GRANT EXECUTE statements:

```
?SECTION "GRANT SALES SPJs"

  GRANT EXECUTE
    ON samdbcat.sales.monthlyorders
    TO PUBLIC;

  GRANT SELECT
    ON TABLE samdbcat.sales.orders
    TO PUBLIC;


?SECTION "GRANT PERSNL SPJs"

  GRANT EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    TO "HR.MGRNA", "HR.MGREU"
    WITH GRANT OPTION;

  GRANT SELECT, UPDATE(salary)
    ON TABLE samdbcat.persnl.employee
    TO "HR.MGRNA", "HR.MGREU"
    WITH GRANT OPTION;
```

To grant privileges on the SPJs, issue the OBEY command in MXCI:

```
OBEY grantprocs.sql "GRANT SALES SPJs";
```

REVOKE EXECUTE Statements in an OBEY Command File

You can use another or the same OBEY command file to revoke privileges on a series of SPJs. For example, the OBEY command file, `revokeprocs.sql`, contains a series of REVOKE EXECUTE statements:

```
?SECTION "REVOKE SALES SPJs"

  REVOKE EXECUTE
    ON PROCEDURE samdbcat.sales.monthlyorders
    FROM PUBLIC;

  REVOKE SELECT
    ON TABLE samdbcat.sales.orders
    FROM PUBLIC;


?SECTION "REVOKE PERSNL SPJs"

  REVOKE GRANT OPTION FOR EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    FROM "HR.MGRNA", "HR.MGREU"
    CASCADE;

  REVOKE EXECUTE
    ON PROCEDURE samdbcat.persnl.adjustsalary
    FROM "HR.MGRNA", "HR.MGREU"
    CASCADE;

  REVOKE SELECT, UPDATE(salary)
    ON TABLE samdbcat.persnl.employee
    FROM "HR.MGRNA", "HR.MGREU"
    CASCADE;
```

To revoke privileges on the SPJs, issue the OBEY command in MXCI:

```
OBEY revokeprocs.sql "REVOKE SALES SPJs";
```

Regenerating CREATE PROCEDURE Statements

Use the SHOWDDL command to regenerate the CREATE PROCEDURE statements of registered SPJs in a new or existing OBEY command file:

1. In an MXCI session, enter the LOG command so that MXCI writes the output of the SHOWDDL command to a log file:

   ```
   LOG createprocs.sql;
   ```

2. In the same MXCI session, enter the SHOWDDL command for specific SPJs in the database:

   ```
   SHOWDDL PROCEDURE samdbcat.sales.dailyorders;
   SHOWDDL PROCEDURE samdbcat.sales.monthlyorders;
   ...
   ```

   For the syntax of the SHOWDDL command, see the *SQL/MX Reference Manual*.

3. To end logging, enter the LOG command without the name of the log file:

```
LOG;
```

4. Edit the log file by removing the SHOWDDL commands and by revising the CREATE PROCEDURE statements if necessary.

   In this example, you must remove the text in **boldface** from the log file. The output of the SHOWDDL command does not exactly match the original CREATE PROCEDURE statement. For more information, see the Output of the SHOWDDL Command (page 93).

```
>>showddl
procedure samdbcat.sales.dailyorders;

CREATE PROCEDURE SAMDBCAT.SALES.DAILYORDERS
  (
    IN DATE
  , OUT NUMBER INTEGER
  )
  EXTERNAL NAME 'Sales.numDailyOrders (java.sql.Date,int[])'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LOCATION \KINGPIN.$TX0115.ZSDX7KT4.N71HTB00
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA
  NOT DETERMINISTIC
  ISOLATE
  ;

--- SQL operation complete.
>>showddl procedure samdbcat.sales.monthlyorders;

CREATE PROCEDURE SAMDBCAT.SALES.MONTHLYORDERS
  (
    IN INTEGER
  , OUT NUMBER INTEGER
  )
  EXTERNAL NAME 'Sales.numMonthlyOrders (int,int[])'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LOCATION \KINGPIN.$TX0115.ZSDX7KT4.G13HVB00
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA
  NOT DETERMINISTIC
  ISOLATE
  ;

--- SQL operation complete.
>>log;
```

As you can with any OBEY command file, you can use the edited log file to re-create the SPJs if necessary. For more information, see CREATE PROCEDURE Statements in an OBEY Command File (page 94).

Backing Up SPJs

Backing up SPJs should be part of a backup and recovery strategy for the entire database. This strategy should involve periodically storing copies of these files in a safe location, such as another disk, magnetic tape, or another system:

- CREATE PROCEDURE statements (or the equivalent SQL/MX metadata)
- Java class or JAR files that contain the Java classes of the SPJs

For comprehensive strategies of backing up and recovering database files, see the SQL/MX Installation and Upgrade Guide and SQL/MX Management Manual.

Using SPJs in a Distributed Database Environment

For an SPJ to operate in a distributed database environment, the catalog in which the SPJ is created must be visible on every remote node that requires access to that catalog. In other words, the catalog of the SPJ must be visible on each node where you want users to be able to call the SPJ.

To make a catalog visible on a remote node, you must register the catalog on the remote node by issuing a REGISTER CATALOG statement on a node where the catalog is visible, such as on the local node. For the syntax of the REGISTER CATALOG statement, see the *SQL/MX Reference Manual*. For information about how to manage a distributed database environment, see the *SQL/MX Installation and Management Guide*.

In a distributed database environment, NonStop SQL/MX does not replicate or distribute the Java class that underlies an SPJ. If you create an SPJ on a local node and want to call that SPJ on a remote node, use one of these approaches:

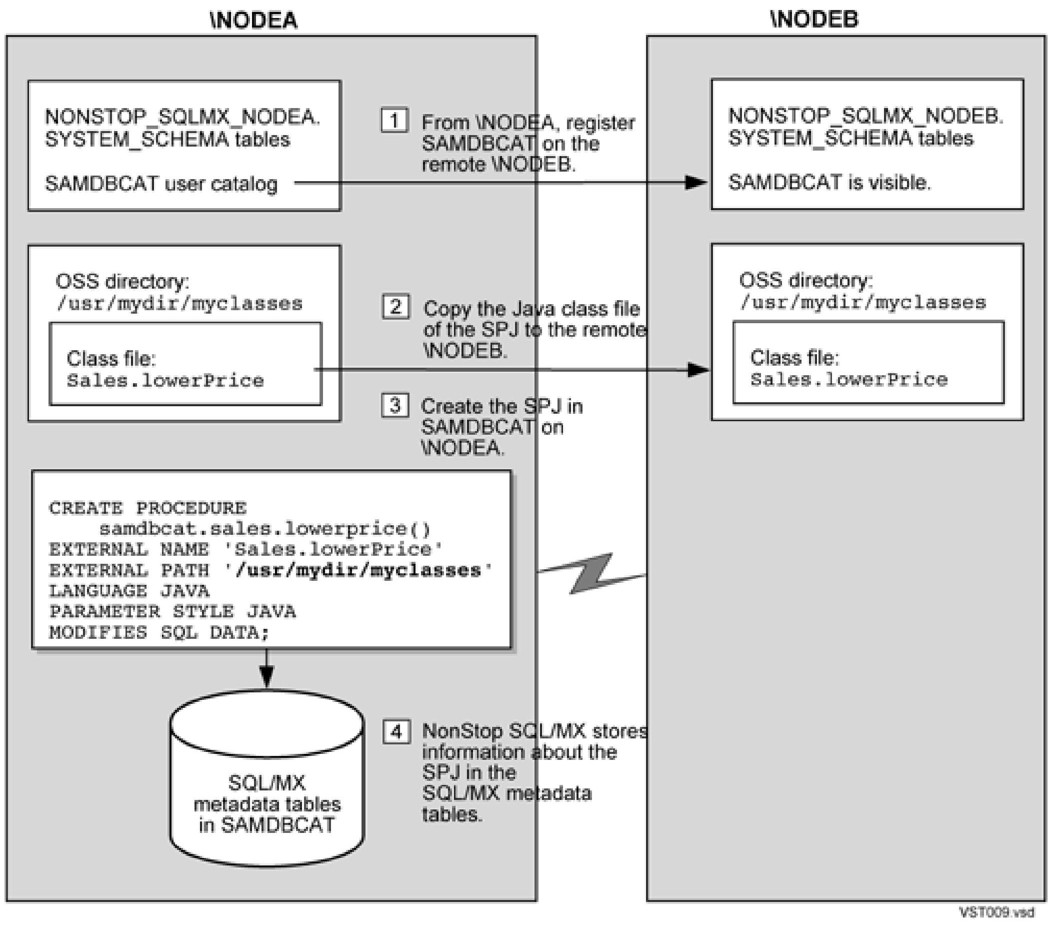
- Copying Java Classes to the Remote Node (page 98)
- Specifying the Node in the External Path (page 99)

Make sure that the application classes on which the SPJ depends are also available in the distributed database environment. For more information, see Distributing Application Classes (page 100).

Copying Java Classes to the Remote Node

Copy all the Java classes required for the SPJ from the local node (for example, `\NODEA`) to the same OSS directory on the remote node (for example, `\NODEB`). By default, only a relative path of the Java class is stored in the SQL/MX system metadata. Therefore, when an application on the remote `\NODEB` calls the SPJ, the JVM looks for the Java class in the OSS directory on `\NODEB`. See Figure 9.

Figure 9 Copying Java Classes to the Remote Node


\NODEA
\NODEB
NONSTOP_SQLMX_NODEA.
SYSTEM_SCHEMA tables
SAMDBCAT user catalog
1 From \NODEA, register SAMDBCAT on the remote \NODEB.
NONSTOP_SQLMX_NODEB.
SYSTEM_SCHEMA tables
SAMDBCAT is visible.
OSS directory: /usr/mydir/myclasses
Class file: Sales.lowerPrice
2 Copy the Java class file of the SPJ to the remote \NODEB.
OSS directory: /usr/mydir/myclasses
Class file: Sales.lowerPrice
3 Create the SPJ in SAMDBCAT on \NODEA.
4 NonStop SQL/MX stores information about the SPJ in the SQL/MX metadata tables.
SQL/MX metadata tables in SAMDBCAT
VST009.vsd

```
CREATE PROCEDURE
    samdbcat.sales.lowerprice()
EXTERNAL NAME 'Sales.lowerPrice'
EXTERNAL PATH '/usr/mydir/myclasses'
LANGUAGE JAVA
PARAMETER STYLE JAVA
MODIFIES SQL DATA;
```

Advantages of Copying the Java Classes

The advantage of this approach is that there are fewer cross-node dependencies. If the network is disconnected or if one node goes down (for example, `\NODEA`), a statically compiled application can still execute the SPJ on the remaining node (for example, `\NODEB`), as long as the procedure label of the SPJ is still available.

The procedure label is available if it is located on the same node (in this case, `\NODEB`) as the calling application or if it is located on a remote node that is still connected and visible (for example, `\NODEC`). If the procedure label is unavailable (for example, if it is on `\NODEA`), the CALL statement that invokes the SPJ will fail. For more information about the procedure label, see Naming the Procedure Label (page 66) TPSEC04.fm Naming the Procedure Label and Showing the Procedure Label (page 91).

Currently, NonStop SQL/MX has limited local node autonomy because it disallows the replication of metadata from one node to another. For more information about the limitations on local node autonomy, see the *SQL/MX Installation* and *Upgrade Guide and SQL/MX Management Manual.*

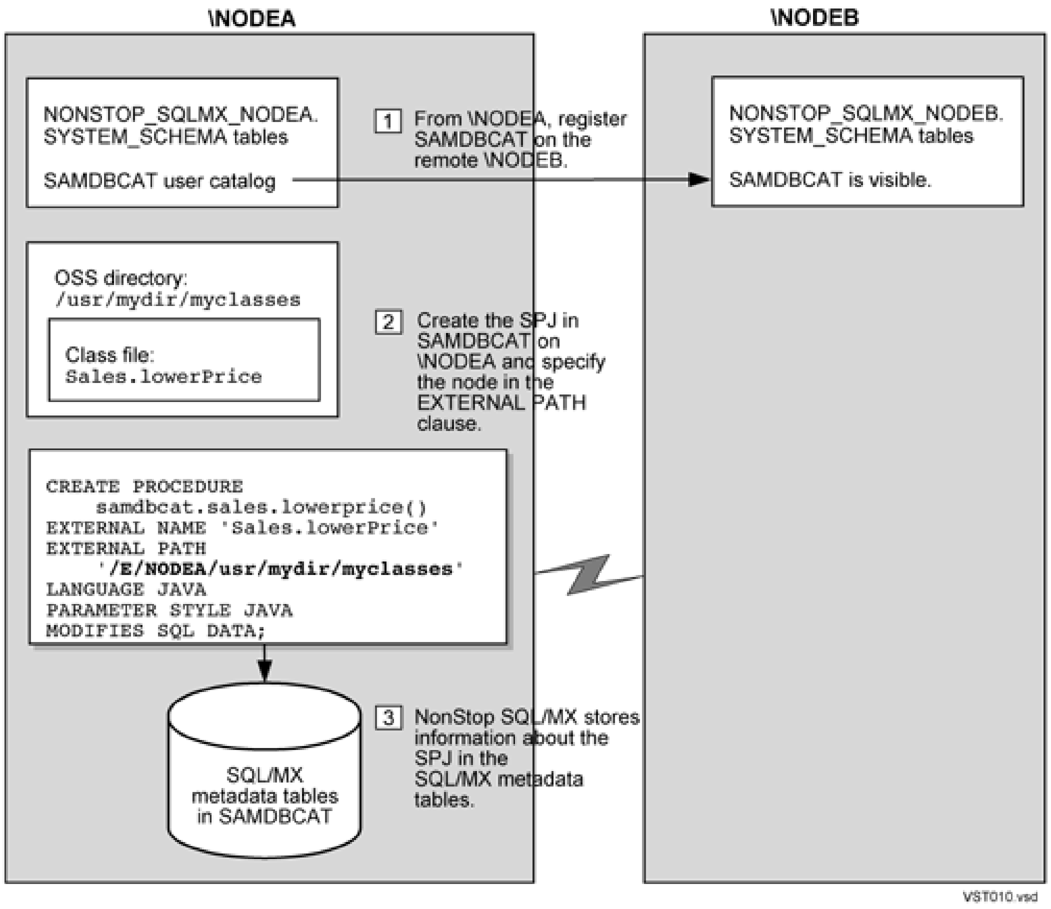
Disadvantages of Copying the Java Classes

The disadvantage of this approach is that the Java classes must be maintained on more than one node. If you change and recompile the Java class on one node, you must do the same on the other node. Otherwise, the SPJ might behave inconsistently in a distributed database environment.

Specifying the Node in the External Path

Register the SPJ with an external path that begins with the name of the node. That is, specify the full path (for example, `/E/NODEA/usr/mydir/myclasses`) in the EXTERNAL PATH clause of the CREATE PROCEDURE statement. The full path of the Java class is stored in the SQL/MX system metadata, as shown in Figure 10.

Figure 10 Specifying the Node in the External Path


\NODEA
\NODEB
NONSTOP_SQLMX_NODEA.
SYSTEM_SCHEMA tables
SAMDBCAT user catalog
1 From \NODEA, register SAMDBCAT on the remote \NODEB.
NONSTOP_SQLMX_NODEB.
SYSTEM_SCHEMA tables
SAMDBCAT is visible.
OSS directory:
/usr/mydir/myclasses
Class file:
Sales.lowerPrice
2 Create the SPJ in SAMDBCAT on \NODEA and specify the node in the EXTERNAL PATH clause.
3 NonStop SQL/MX stores information about the SPJ in the SQL/MX metadata tables.
SQL/MX metadata tables in SAMDBCAT
VST010.vsd

```
CREATE PROCEDURE
    samdbcat.sales.lowerprice()
EXTERNAL NAME 'Sales.lowerPrice'
EXTERNAL PATH
    '/E/NODEA/usr/mydir/myclasses'
LANGUAGE JAVA
PARAMETER STYLE JAVA
MODIFIES SQL DATA;
```

When an application on the remote `\NODEB` calls the SPJ, the JVM looks for the Java class in the `/E/NODEA/usr/mydir/myclasses` directory on `\NODEA`. Regardless of where you issue the CALL statement, the JVM in the SPJ environment always tries to load the Java class from the OSS directory on the node that you specify in the EXTERNAL PATH clause.

Advantages of Specifying the Node

The Java classes exist in only one location, which eliminates the need to maintain them on separate nodes.

Disadvantages of Specifying the Node

By using this approach, you sacrifice local node autonomy. For example, applications running on `\NODEB` rely on the Java class stored on `\NODEA` to function properly. If `\NODEB` becomes disconnected from `\NODEA`, an application on `\NODEB` that calls the SPJ on `\NODEA` will not work.

This approach also involves the overhead of loading Java classes from a remote node instead of from the local node.

Distributing Application Classes

If an SPJ method in a distributed database environment relies on application classes outside the external path, you must either copy these classes to the same OSS directory on the remote node or include the full path of these classes in the class path. For more information about how to set the class path, see Setting the Class Path (page 43).

7 Performance and Troubleshooting

This section describes how to improve and monitor the performance of SPJs in NonStop SQL/MX, provides guidelines for troubleshooting common problems, and covers these topics:

- Troubleshooting SPJ Problems (page 101)
- Performance Tips (page 101)
- Displaying an Execution Plan for a CALL Statement (page 102)
- Displaying the Shape of a CALL Statement (page 103)

Troubleshooting SPJ Problems

To resolve problems that occur when you register or invoke an SPJ, follow these guidelines:

- Note the SQLCODE or SQLSTATE value of the error messages and locate the information in the *SQL/MX Messages Manual*, which provides cause, effect, and recovery information for all SQL/MX errors.
- Check that the user has the appropriate permissions to create or call the SPJ. See Required Privileges for Creating an SPJ (page 59), Required Privileges for Calling an SPJ (page 71), and Showing Privileges on the SPJs (page 89).
- Check the code of the SPJ method. See Chapter 3: Writing SPJ Methods. Fix any problems.
- If you successfully compiled and registered the SPJ but are receiving errors when calling the SPJ, check that the output parameters in the Java method are specified as arrays. See Returning Output Values From the Java Method (page 50).
- Check the syntax of the CREATE PROCEDURE statement by using the SHOWDDL command. See Showing the Syntax of an SPJ (page 92). Compare the CREATE PROCEDURE syntax with the SPJ method. See Using the CREATE PROCEDURE Statement (page 59). Fix any problems.
- Check the syntax of the CALL statement in the application. See Chapter 5: Invoking SPJs in NonStop SQL/MX. Fix any problems.
- If the SQL/MX syntax and SPJ code are correct, check the configuration of the SPJ environment. Verify the JREHOME, JDBC/MX location, and class path settings. Reconfigure the JVM startup options (that is, UDR_JAVA_OPTIONS settings) to determine if the problem is related to the SPJ environment. See Controlling JVM Startup Options (page 36).
- If you receive an error describing a Java security-related problem, check the UDR_JAVA_OPTIONS setting and the permissions in the specified policy file. Verify that all the Java classes have required permissions. See Establishing Java Security (page 46).
- To identify Java-related errors, execute the SPJ method outside the database by invoking the Java method directly in a Java application that you run on the command line. For more information, see the *NonStop Server for Java Programmer's Reference*.

Performance Tips

To ensure the optimal performance of SPJs in NonStop SQL/MX:

- Design your Java classes and directory structure so that fewer SPJ methods in different external paths rely on the same application class. This approach conserves system resources and memory. For more information, see Updating the Java Class Files of a Referenced Application Class (page 30).
- Use UDR_JAVA_OPTIONS settings to manage multiple SPJ environments to support parallelism of CALL statements issued from a multithreaded application. For more information, see Multithreading in an SPJ Environment (page 27).

- Avoid using too many UDR_JAVA_OPTIONS settings in applications that call SPJs. For more information, see Using Multiple UDR_JAVA_OPTIONS Settings in One Application (page 39).
- Avoid nesting CALL statements in an SPJ method, which wastes resources and diminishes performance. For more information, see Nested Java Method Invocations (page 53).
- Use connection pooling and explicitly close each `java.sql.Connection` object when it is no longer needed instead of relying on garbage collection. For more information, see Use of java.sql.Connection Objects (page 53).

Displaying an Execution Plan for a CALL Statement

An execution plan reveals how a CALL statement was optimized. You can display all or part of the execution plan for a CALL statement by using the DISPLAY_EXPLAIN command or the EXPLAIN function.

Using the DISPLAY_EXPLAIN Command

Suppose that you want to display the execution plan for this CALL statement:

```
CALL samdbcat.persnl.adjustsalary(202, 5.5, ?);
```

Enter this DISPLAY_EXPLAIN command in an MXCI session:

```
DISPLAY_EXPLAIN CALL samdbcat.persnl.adjustsalary(202, 5.5, ?);
```

The DISPLAY_EXPLAIN command generates and displays all the columns of the result table of the EXPLAIN function. For more information, see the *SQL/MX Query Guide*.

Using the EXPLAIN Function

You can also prepare the CALL statement and select specific columns from the result table of the EXPLAIN function, as shown:

```
PREPARE call_spj
  FROM CALL samdbcat.persnl.adjustsalary(202, 5.5, ?);

SELECT SUBSTRING(OPERATOR,1,8) AS "OPERATOR", OPERATOR_COST,
       SUBSTRING(DESCRIPTION,1,500) AS "DESCRIPTION"
  FROM TABLE (EXPLAIN(NULL, 'CALL_SPJ'));
```

The SELECT statement displays this output:

```
OPERATOR  OPERATOR_COST    DESCRIPTION
--------  ---------------  -----------------------------------------------

CALL     1.3930000E-005   input_values: cast(202), cast(cast((cast(5.5)/
cast(10)))) parameter_modes: I I O  routine_name:
SAMDBCAT.PERSNL.ADJUSTSALARY routine_label:
\KINGPIN.$TX0115.ZSDPK4GV.Q85DXB00 sql_access_mode: MODIFIES SQL DATA
external_name: adjustSalary external_path: /usr/mydir/myclasses
external_file: Payroll signature:
(Ljava/math/BigDecimal;D[Ljava/math/BigDecimal;)V language: Java
runtime_options: OFF runtime_option_delimiters: ' '

ROOT    5.8506000E-008   select_list: NUMERIC(8,2) SIGNED input_variables:
? statement_index: 0 statement: CALL samdbcat.persnl.adjustsalary(202,
5.5, ?); return
```

For a CALL statement, the OPERATOR column of the result table contains a row named `CALL`. The DESCRIPTION column contains special token pairs for the CALL operator. For more information, see the *SQL/MX Query Guide*.

Using the EXPLAIN function with SPJ RS

You can also prepare the CALL statement and return result sets from the result table of the EXPLAIN function, as shown:

```
PREPARE S
FROM CALL samdbcat.sales.order_summary(?, ?);

SELECT DESCRIPTION FROM TABLE(EXPLAIN(NULL, 'S')) WHERE OPERATOR
= 'CALL';
```

The SELECT statement displays the following output:

```

DESCRIPTION
--------------------------------------------------------------------------
--
parameter_modes: I O routine_name: SAMDBCAT.SALES.ORDER_SUMMARY
routine_label: \ALPINE.$SYSTEM.ZSDCR2C6.L1Z7NW00
sql_access_mode: READS SQL DATA external_name: orderSummary2
external_path: /usr/mydir/myclasses external_file: rs
signature:
(Ljava/lang/String;[J[Ljava/sql/ResultSet;[Ljava/sql/ResultSet;)V
language: Java runtime_options: OFF runtime_option_delimiters: ' '
max_results: 2
--- 1 row(s) selected.
```

The output includes a new entry max_results, which displays the maximum number of result sets that can be returned by this procedure. The value for max-results is set by the DYNAMIC RESULTS option in the CREATE PROCEDURE statement.

NOTE: The max_results option is displayed only on systems running J06.05 and later J-series RVUs or H06.16 and later H-series RVUs.

Displaying the Shape of a CALL Statement

The SHOWSHAPE command displays the shape for a given DML statement, such as a CALL statement, in an MXCI session. The result can be used at a later time to force the SQL/MX compiler to choose a particular execution plan (or to force the same execution plan for the statement).

CALL Statements Without a Shape

A CALL statement that does not contain a subquery as one of its SQL parameter arguments does not have a shape. For example, this SHOWSHAPE command is for a CALL statement without subquery parameters:

```
SHOWSHAPE CALL samdbcat.persnl.adjustsalary(202,5.5,?);
```

Because this CALL statement does not have a shape, the SHOWSHAPE command returns this output:

```
control query shape anything;
--- SQL operation complete.
```

CALL Statements With a Shape

If a CALL statement contains one or more subqueries as its SQL parameter arguments, the SHOWSHAPE command generates a shape that describes the entire SQL statement. For example, this SHOWSHAPE command is for a CALL statement with a subquery parameter:

```
SHOWSHAPE CALL samdbcat.sales.totalprice((SELECT qty_ordered
                                FROM samdbcat.sales.odetail
                                  WHERE ordernum = 100210
```

```
                                        AND partnum = 5100),
                                   'standard', ?param);
```

The generated shape describes all subquery trees and contains an ANYTHING token to represent the CALL statement, as shown:

```
control query shape nested_join(nested_join(sort_groupby(partition_access(
scan(path 'SAMDBCAT.SALES.ODETAIL', forward, blocks_per_access 1
, mdam off))),tuple),anything);

--- SQL operation complete.
```

If the result of the SHOWSHAPE command is undesirable, you can use the CONTROL QUERY SHAPE statement to force the execution plan. First generate the result of the EXPLAIN function for a prepared CALL statement, and then modify the operator tree for the execution plan by using CONTROL QUERY SHAPE.

For more information about the SHOWSHAPE command and the CONTROL QUERY SHAPE statement, see the *SQL/MX Reference Manual*. For information about forcing execution plans, see the *SQL/MX Query Guide*.

A Sample SPJs

This appendix presents the SPJs that are shown in examples throughout this manual. For descriptions of the SPJ methods and the CREATE PROCEDURE statements that register those methods in an SQL/MX database, see:

- Class Files and Java Methods (page 106)
- The createprocs.sql File (page 111)
- Examples of the Sample SPJs (page 113)

Using Sample SPJs

The SPJ methods are coded in Java source files, which are located in a JAR file named `SampleSPJs.jar` in the NonStop Technical Library (NTL).

Installing the SampleSPJs.jar File

1. In NTL, navigate to the H06.04G06.23 or later RVU.
2. From the Categories list, select **Sample Programs**.
3. Under Publications, select **SQL/MX Stored Procedures in Java (SPJs)**.
4. Follow the instructions in NTL for downloading the `SampleSPJs.jar` file to your PC workstation.
5. On your PC, use WinZip to open the `SampleSPJs.jar` file and view the `README.txt` file, which explains how to use `SampleSPJs.jar`.

To use the source files in `SampleSPJs.jar`, place the JAR file in the desired OSS directory, extract the source files, and compile them by using the Java programming language compiler. See Figure 11 (page 106).

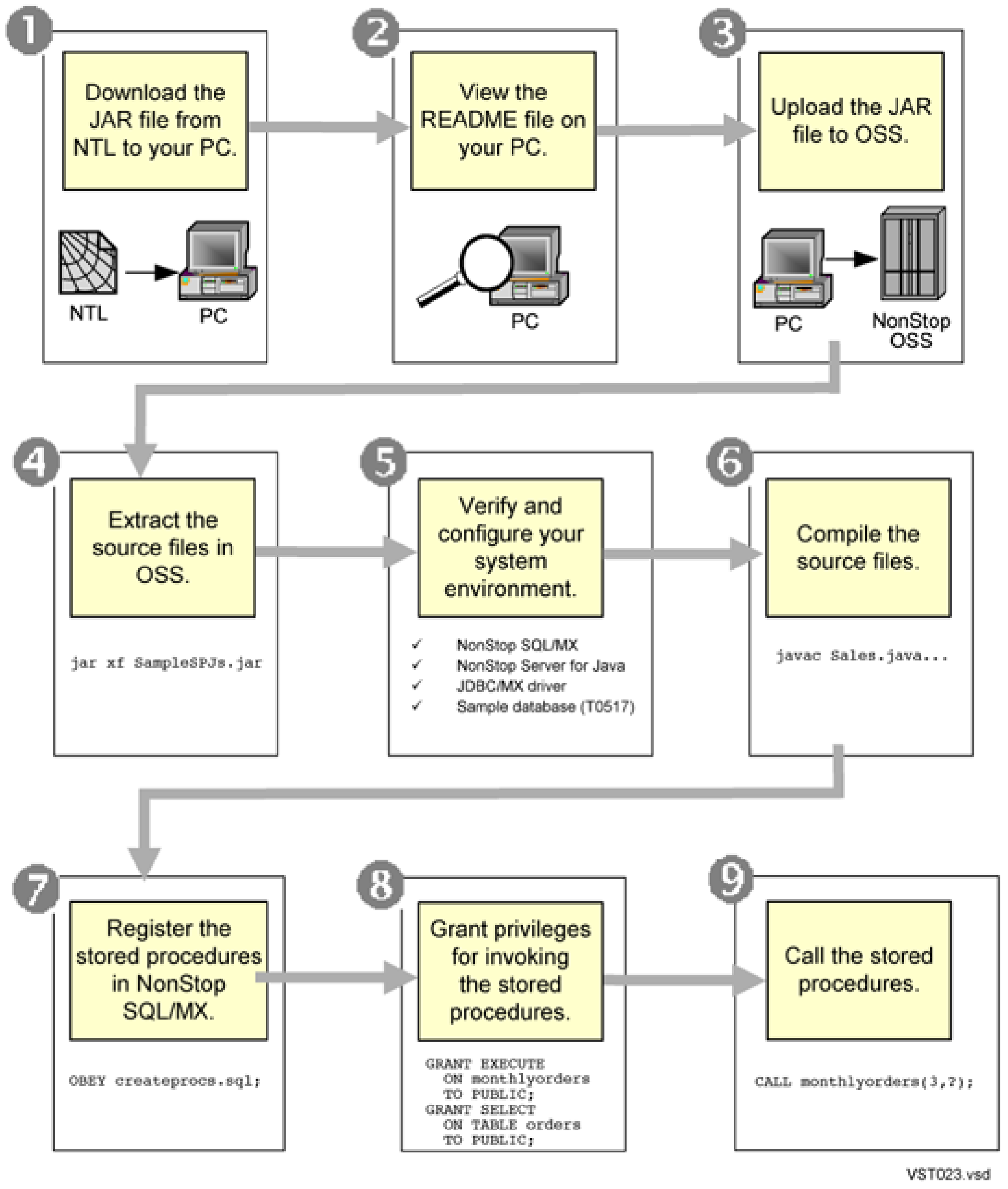
Installing the SQL/MX Sample Database

The SPJs rely on the SQL/MX sample database (product T0517) to run properly. All the sample SPJs can query both SQL/MP and SQL/MX tables of the SQL/MX sample database. To query SQL/MP tables, use the NonStop SQL/MX Release 1.8 sample database. To query SQL/MX tables, use the NonStop SQL/MX Release 2.x sample database.

NOTE: The SQL/MX Release 2.x sample database uses SQL/MX format tables. To install the sample database, you must have a license to use SQL/MX DDL statements. To acquire this license, purchase product T0394. Without this product, you cannot install the sample database. An error message informs you that the system is not licensed.

To install the sample database, see the *SQL/MX Quick Start*. For information about the schema and tables of the sample database, see the *SQL/MX Reference Manual*.

Figure 11 Steps for Using Sample SPJs


1
Download the JAR file from NTL to your PC.
NTL
PC
2
View the README file on your PC.
PC
3
Upload the JAR file to OSS.
PC
NonStop OSS
4
Extract the source files in OSS.
jar xf SampleSPJs.jar
5
Verify and configure your system environment.
✓ NonStop SQL/MX
✓ NonStop Server for Java
✓ JDBC/MX driver
✓ Sample database (T0517)
6
Compile the source files.
javac Sales.java...
7
Register the stored procedures in NonStop SQL/MX.
OBEY createprocs.sql;
8
Grant privileges for invoking the stored procedures.
GRANT EXECUTE ON monthlyorders TO PUBLIC;
GRANT SELECT ON TABLE orders TO PUBLIC;
9
Call the stored procedures.
CALL monthlyorders(3,?);
VST023.vsd

For details of these steps, see the README file in `SampleSPJs.jar`.

Class Files and Java Methods

The SPJ methods are stored in these class files:

- Sales Class (page 107)
- Payroll Class (page 109)
- Inventory Class (page 110)

These class files are coded as Java source files and use JDBC/MX to access a NonStop SQL database. For information about JDBC/MX, see the *JDBC Driver for SQL/MX Programmer's Reference*.

Sales Class

The `Sales` class contains these SPJ methods, which are useful for tracking orders and managing sales:

- The `lowerPrice()` method determines which items are selling poorly (that is, have less than 50 orders) and lowers the price of these items in the database by 10 percent.
- The `numDailyOrders()` method accepts a date and returns the number of orders on that date to an output parameter.
- The `numMonthlyOrders()` method accepts an integer representing the month and returns the number of orders during that month to an output parameter.
- The `totalPrice()` method accepts the quantity, shipping speed, and price of an item, calculates the total price, including tax and shipping charges, and returns the total price to an input/output parameter.

The `Sales.java` source file in `SampleSPJs.jar` contains the code shown in Example 1 (page 108).

Example 1 Sales.java—The Sales Class

```
import java.sql.*;
import java.math.*;

public class Sales
{
    public static void lowerPrice()
        throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement getParts =
            conn.prepareStatement("SELECT p.partnum, " +
                                  "  SUM(qty_ordered) AS qtyOrdered" +
                                  "FROM samdbcat.sales.parts p "+
                                  "LEFT JOIN samdbcat.sales.odetailo " +
                                  "  ON p.partnum = o.partnum "+
                                  "GROUP BY p.partnum");

        PreparedStatement updateParts =
            conn.prepareStatement("UPDATE samdbcat.sales.parts "+
                                  "SET price = price * 0.9 " +
                                  "WHERE partnum = ?");

        ResultSet rs = getParts.executeQuery();
        while (rs.next())
        {
            BigDecimal qtyOrdered = rs.getBigDecimal(2);
            if ((qtyOrdered == null) || (qtyOrdered.intValue() <50))
            {
                BigDecimal partnum = rs.getBigDecimal(1);
                updateParts.setBigDecimal(1, partnum);
                updateParts.executeUpdate();
            }
        }
        rs.close();

        conn.close();
    }
    public static void numDailyOrders(Date date,
                                          int[] numOrders)
        throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement getNumOrders =
            conn.prepareStatement("SELECT COUNT(order_date) " +
                                  "FROM samdbcat.sales.orders " +
                                  "WHERE order_date = ?");


        getNumOrders.setDate(1, date);
        ResultSet rs = getNumOrders.executeQuery();
        rs.next();
        numOrders[0] = rs.getInt(1);
        rs.close();

        conn.close();
    }


    public static void
numMonthlyOrders(int month,
                                       int[] numOrders)
        throws SQLException
    {
        if ( month < 1 || month > 12 )
        {
            throw new
                SQLException ("Invalid value for month. " +
                              "Retry the CALL statement " +
                              "using a number from 1 to 12 " +
                              "to represent the month.", "38001");
        }

        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement getNumOrders =
            conn.prepareStatement("SELECT COUNT(month(order_date))" +
                                  "FROM samdbcat.sales.orders "+
                                  "WHERE month(order_date) = ?");

        getNumOrders.setInt(1, month);
        ResultSet rs = getNumOrders.executeQuery();
        rs.next();
        numOrders[0] = rs.getInt(1);
        rs.close();

        conn.close();
    }
    public static voidtotalPrice(BigDecimal qtyOrdered,
                                 String shippingSpeed,
                                 BigDecimal[] price)
        throws SQLException
    {
        BigDecimal shipcharge = new BigDecimal(0);

        if (shippingSpeed.equals("economy"))
        {
            shipcharge = new BigDecimal(1.95);
        }
        else if (shippingSpeed.equals("standard"))
        {
            shipcharge = new BigDecimal(4.99);
        }
        else if (shippingSpeed.equals("nextday"))
        {
            shipcharge = new BigDecimal(14.99);
        }
        else
        {
            throw new
                SQLException ("Invalid value for shipping speed." +
                              "Retry the CALL statement using "+
                              "'economy' for 7 to 9 days," +
                              "'standard' for 3 to 5 days, or "+
                              "'nextday' for one day.", "38002");
        }

        BigDecimal subtotal = price[0].multiply(qtyOrdered);

        BigDecimal tax = new BigDecimal(0.0825);
        BigDecimal taxcharge = subtotal.multiply(tax);

        BigDecimal charges = taxcharge.add(shipcharge);

        price[0] = subtotal.add(charges);
    }
}
```

Payroll Class

The `Payroll` class contains these SPJ methods, which are useful for managing personnel data:

- The `adjustSalary()` method accepts an employee number and a percentage value and updates the employee's salary in the database based on this value. This method also returns the updated salary to an output parameter.
- The `employeeJob()` method accepts an employee number and returns a job code or null value to an output parameter.

The `Payroll.java` source file in `SampleSPJs.jar` contains the code shown in Example 2.

Example 2 Payroll.java-The PayrollClass

```
import java.sql.*;
import java.math.*;

public class Payroll
{
    public static void adjustSalary(BigDecimal empNum,
                                    double percent,
                                    BigDecimal[] newSalary)
        throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement setSalary =
            conn.prepareStatement("UPDATE samdbcat.persnl.employee" +
                                  "SET salary = salary * (1 + (?/ 100)) " +
                                  "WHERE empnum = ?");

        PreparedStatement getSalary =
            conn.prepareStatement("SELECT salary " +
                                  "FROM samdbcat.persnl.employee" +
                                  "WHERE empnum = ?");

        setSalary.setDouble(1, percent);
        setSalary.setBigDecimal(2, empNum);
        setSalary.executeUpdate();

        getSalary.setBigDecimal(1, empNum);
        ResultSet rs = getSalary.executeQuery();
        rs.next();
        newSalary[0] = rs.getBigDecimal(1);
        rs.close();

        conn.close();
    }

    public static void employeeJob(int empNum,
                                   java.lang.Integer[] jobCode)
        throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement getJobcode =
            conn.prepareStatement("SELECT jobcode " +
                                  "FROM samdbcat.persnl.employee" +
                                  "WHERE empnum = ?");

        getJobcode.setInt(1, empNum);
        ResultSet rs = getJobcode.executeQuery();
        rs.next();
        int num = rs.getInt(1);
        if (rs.wasNull())
            jobCode[0] = null;
        else
            jobCode[0] = new Integer(num);
        rs.close();

        conn.close();
    }
}
```

Inventory Class

The `Inventory` class contains these SPJ methods, which are useful for tracking parts and suppliers:

- The `supplierInfo()` method accepts a supplier number and returns the supplier's name, street, city, state, and post code to separate output parameters.
- The `supplyQuantities()` method returns the average, minimum, and maximum quantities of available parts in inventory to separate output parameters.

The `Inventory.java` source file in `SampleSPJs.jar` contains the code shown in Example 3.

Example 3 Inventory.java-The Inventory Class

```
import java.sql.*;
import java.math.*;

public class Inventory
{
    public static void supplierInfo(BigDecimal suppNum,
                                    String[] suppName,
                                    String[] streetAddr,
                                    String[] cityName,
                                    String[] stateName,
                                    String[] postCode)
        throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement getSupplier =
            conn.prepareStatement("SELECT suppname, street, city," +
                                  "  state, postcode " +
                                  "FROM samdbcat.invent.supplier" +
                                  "WHERE suppnum = ?");

        getSupplier.setBigDecimal(1, suppNum);
        ResultSet rs = getSupplier.executeQuery();
        rs.next();
        suppName[0] = rs.getString(1);
        streetAddr[0] = rs.getString(2);
        cityName[0] = rs.getString(3);
        stateName[0] = rs.getString(4);
        postCode[0] = rs.getString(5);
        rs.close();

        conn.close();
    }

    public static void supplyQuantities(int[] avgQty,
                                        int[] minQty,
                                        int[] maxQty)
        throws SQLException
    {
        Connection conn = DriverManager.getConnection("jdbc:default:connection");

        PreparedStatement getQty =
            conn.prepareStatement("SELECT AVG(qty_on_hand), " +
                                  "  MIN(qty_on_hand), " +
                                  "  MAX(qty_on_hand) " +
                                  "FROM samdbcat.invent.partloc");

        ResultSet rs = getQty.executeQuery();
        rs.next();
        avgQty[0] = rs.getInt(1);
        minQty[0] = rs.getInt(2);
        maxQty[0] = rs.getInt(3);
        rs.close();

        conn.close();
    }
}
```

The createprocs.sql File

CREATE PROCEDURE statements register SPJ methods in an SQL/MX database. Before registering the methods described in Class Files and Java Methods (page 106), you must compile the Java source files into class files. See Compiling Java Classes (page 58).

An OBEY command file named `createprocs.sql` in `SampleSPJs.jar` contains the CREATE PROCEDURE statements shown in Example 4.

Example 4 createprocs.sql-An OBEY Command File

```
CREATE PROCEDURE samdbcat.sales.lowerprice()
  EXTERNAL NAME 'Sales.lowerPrice'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA;

CREATE PROCEDURE samdbcat.sales.dailyorders(IN DATE, OUT number INT)
  EXTERNAL NAME 'Sales.numDailyOrders'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;

CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT, OUT number INT)
  EXTERNAL NAME 'Sales.numMonthlyOrders'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;

CREATE PROCEDURE samdbcat.sales.totalprice(IN qty NUMERIC(18),
                                           IN rate VARCHAR(10),
                                           INOUT price NUMERIC(18,2))
  EXTERNAL NAME 'Sales.totalPrice'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;

CREATE PROCEDURE samdbcat.persnl.adjustsalary(IN empnum NUMERIC(4),
                                              IN percent FLOAT,
                                              OUT newsalary NUMERIC(8,2))
  EXTERNAL NAME 'Payroll.adjustSalary'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA;


CREATE PROCEDURE samdbcat.persnl.employeejob(IN empnum INT,
                                             OUT jobcode INT)
  EXTERNAL NAME 'Payroll.employeeJob(int, java.lang.Integer[])'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;

CREATE PROCEDURE samdbcat.invent.supplierinfo(INempnum NUMERIC(4),
                                              OUT suppname CHAR(18),
                                              OUT address CHAR(22),
                                              OUT city CHAR(14),
                                              OUT state CHAR(12),
                                              OUT zipcode CHAR(10))
  EXTERNAL NAME 'Inventory.supplierInfo'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;

CREATE PROCEDURE samdbcat.invent.supplynumbers(OUT avrg INT,
                                               OUT minm INT,
                                                    OUT maxm INT)
  EXTERNAL NAME 'Inventory.supplyQuantities'
  EXTERNAL PATH '<mydir>'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

Before executing the OBEY command file, `createprocs.sql`, change the external paths from *`<dir>`* to the OSS directories that contain the SPJ class files.

The catalog and schemas of stored procedures must exist before you issue CREATE PROCEDURE statements. The CREATE PROCEDURE statements in `createprocs.sql` use the SAMDBCAT catalog and the SALES, PERSNL, and INVENT schemas, which are part of the SQL/MX sample database. To install the sample database, see the *SQL/MX Quick Start*.

To execute the CREATE PROCEDURE statements in `createprocs.sql`, you must be either the schema owner (which you are if you install the sample database yourself) or the super ID.

Examples of the Sample SPJs

These examples show each SPJ method, the CREATE PROCEDURE statement that registers the SPJ, the CALL statement that invokes the SPJ, and the output in MXCI:

- LOWERPRICE Stored Procedure (page 113)
- DAILYORDERS Stored Procedure (page 115)
- MONTHLYORDERS Stored Procedure (page 115)
- TOTALPRICE Stored Procedure (page 116)
- ADJUSTSALARY Stored Procedure (page 117)
- EMPLOYEEJOB Stored Procedure (page 118)
- SUPPLIERINFO Stored Procedure (page 119)
- SUPPLYNUMBERS Stored Procedure (page 120)

LOWERPRICE Stored Procedure

Java Method

```
public static void lowerPrice()
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getParts =
        conn.prepareStatement("SELECT p.partnum, " +
                              "  SUM(qty_ordered) AS qtyOrdered" +
                              "FROM samdbcat.sales.parts p " +
                              "LEFT JOIN samdbcat.sales.odetailo " +
                              "  ON p.partnum = o.partnum " +
                              "GROUP BY p.partnum");

    PreparedStatement updateParts =
        conn.prepareStatement("UPDATE samdbcat.sales.parts " +
                              "SET price = price * 0.9 " +
                              "WHERE partnum = ?");

    ResultSet rs = getParts.executeQuery();
    while (rs.next())
    {
        BigDecimal qtyOrdered = rs.getBigDecimal(2);
        if ((qtyOrdered == null) || (qtyOrdered.intValue() <50))
        {
            BigDecimal partnum = rs.getBigDecimal(1);
            updateParts.setBigDecimal(1, partnum);
            updateParts.executeUpdate();
        }
    }
    rs.close();

    conn.close();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.sales.lowerprice()
  EXTERNAL NAME 'Sales.lowerPrice'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the LOWERPRICE procedure in MXCI:

```
CALL samdbcat.sales.lowerprice();
```

To view the prices and quantities of items in the database with 50 or fewer orders, issue this query before and after calling the LOWERPRICE procedure:

```
SELECT *
  FROM
   (SELECT p.partnum, SUM(qty_ordered) AS qtyOrdered, p.price
      FROM samdbcat.sales.parts p
        LEFT OUTER JOIN samdbcat.sales.odetail o
      ON p.partnum = o.partnum
      GROUP BY p.partnum, p.price) AS allparts
  WHERE qtyOrdered < 51;
```

The LOWERPRICE procedure lowers the price of items with 50 or fewer orders by 10 percent in the database. For example, part number 3103, the LASER PRINTER, X1, has 40 orders and a price of 4200.00:

```
PARTNUM  QTYORDERED             PRICE
-------  --------------------  ------------

    212                    20       2500.00
    244                    47       3000.00
    255                    38       4000.00
   2002                    46       1500.00
    405                    18        795.00
   3103                    40       4200.00
   3201                     6        525.00
   3205                    38        625.00
   3210                     7        715.00
...
```

The invocation of LOWERPRICE lowers the price of this item from 4200.00 to 3780.00:

```
PARTNUM  QTYORDERED             PRICE
-------  --------------------  ------------

    212                    20       2250.00
    244                    47       2700.00
    255                    38       3600.00
   2002                    46       1350.00
   2405                    18        715.50
   3103                    40       3780.00
   3201                     6        472.50
   3205                    38        562.50
   3210                     7        643.50
...
```

DAILYORDERS Stored Procedure

Java Method

```
public static void numDailyOrders(Date date,
                                  int[] numOrders)
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getNumOrders =
        conn.prepareStatement("SELECT COUNT(order_date) " +
                              "FROM samdbcat.sales.orders " +
                              "WHERE order_date = ?");


    getNumOrders.setDate(1, date);
    ResultSet rs = getNumOrders.executeQuery();
    rs.next();
    numOrders[0] = rs.getInt(1);
    rs.close();

    conn.close();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.sales.dailyorders(IN DATE, OUT number INT)
  EXTERNAL NAME 'Sales.numDailyOrders'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the DAILYORDERS procedure in MXCI:

```
CALL samdbcat.sales.dailyorders(DATE '2003-03-19', ?);
```

The DAILYORDERS procedure determines the total number of orders on a specified date and returns this output in MXCI:

```
NUMBER
-----------

          2

--- SQL operation complete.
```

On March 19, 2003, there were two orders.

MONTHLYORDERS Stored Procedure

Java Method

```
public static void numMonthlyOrders(int month,
                                    int[] numOrders)
    throws SQLException
{
    if ( month < 1 || month > 12 )

        throw new
            SQLException ("Invalid value for month. " +
                          "Retry the CALL statement " +
                          "using a number from 1 to 12 " +
                          "to represent the month.", "38001" );
    }

    Connection conn = DriverManager.getConnection("jdbc:default:connection");
```

```
        PreparedStatement getNumOrders =
            conn.prepareStatement("SELECT COUNT(month(order_date)) "+
                                  "FROM samdbcat.sales.orders " +
                                  "WHERE month(order_date) = ?");

        getNumOrders.setInt(1, month);
        ResultSet rs = getNumOrders.executeQuery();
        rs.next();
        numOrders[0] = rs.getInt(1);
        rs.close();

        conn.close();
    }
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.sales.monthlyorders(IN INT, OUT number INT)
  EXTERNAL NAME 'Sales.numMonthlyOrders'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the MONTHLYORDERS procedure in MXCI:

```
CALL samdbcat.sales.monthlyorders(3,?);
```

The MONTHLYORDERS procedure determines the total number of orders during a specified month and returns this output in MXCI:

```
NUMBER
-----------

          4

--- SQL operation complete.
```

In March, there were four orders.

TOTALPRICE Stored Procedure

Java Method

```
public static void totalPrice(BigDecimal qtyOrdered,
                              String shippingSpeed,
                              BigDecimal[] price)
    throws SQLException
{
    BigDecimal shipcharge = new BigDecimal(0);

    if (shippingSpeed.equals("economy"))

        shipcharge = new BigDecimal(1.95);
    }
    else if (shippingSpeed.equals("standard"))

        shipcharge = new BigDecimal(4.99);
    }
    else if (shippingSpeed.equals("nextday"))

        shipcharge = new BigDecimal(14.99);
    }
    else
    {
        throw new
            SQLException ("Invalid value for shipping speed. " +
                          "Retry the CALL statement using " +
                              "'economy' for 7 to 9 days," +
                              "'standard' for 3 to 5 days, or " +
                              "'nextday' for one day.", "38002" );


        BigDecimal subtotal = price[0].multiply(qtyOrdered);

        BigDecimal tax = new BigDecimal(0.0825);
        BigDecimal taxcharge = subtotal.multiply(tax);

        BigDecimal charges = taxcharge.add(shipcharge);

        price[0] = subtotal.add(charges);
    }
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.sales.totalprice(IN qty NUMERIC(18),
                                           IN rate VARCHAR(10),
                                           INOUT price NUMERIC(18,2))
  EXTERNAL NAME 'Sales.totalPrice'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the TOTALPRICE procedure in MXCI:

```
SET PARAM ?p 10; CALL samdbcat.sales.totalprice(23, 'standard', ?p);
```

The TOTALPRICE procedure calculates the total price of a purchase and returns this output in MXCI:

```
PRICE
---------------------

               253.96

--- SQL operation complete.
```

The total price of 23 items, which cost $10 each and which are shipped at the standard rate, is $253.96, including sales tax.

ADJUSTSALARY Stored Procedure

Java Method

```
public static void adjustSalary(BigDecimal empNum,
                                double percent,
                                BigDecimal[] newSalary)
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement setSalary =
        conn.prepareStatement("UPDATE samdbcat.persnl.employee "+
                              "SET salary = salary * (1 + (? / 100))" +
                              "WHERE empnum = ?");

    PreparedStatement getSalary =
        conn.prepareStatement("SELECT salary " +
                              "FROM samdbcat.persnl.employee " +
                              "WHERE empnum = ?");

    setSalary.setDouble(1, percent);
    setSalary.setBigDecimal(2, empNum);
    setSalary.executeUpdate();

    getSalary.setBigDecimal(1, empNum);
    ResultSet rs = getSalary.executeQuery();
    rs.next();
    newSalary[0] = rs.getBigDecimal(1);
    rs.close();

    conn.close();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.persnl.adjustsalary(IN empnum NUMERIC(4),
                                              IN percent FLOAT,
                                              OUT newsalary NUMERIC(8,2))
  EXTERNAL NAME 'Payroll.adjustSalary'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  MODIFIES SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the ADJUSTSALARY procedure in MXCI:

```
CALL samdbcat.persnl.adjustsalary(29, 2.5, ?);
```

The ADJUSTSALARY procedure updates the salary of employee number 29 by 2.5 percent and returns this output in MXCI:

```
NEWSALARY
------------

   139400.00

--- SQL operation complete.
```

The salary of employee number 29 was originally $136,000.00 and became $139,400.00 after the invocation of ADJUSTSALARY.

EMPLOYEEJOB Stored Procedure

Java Method

```
public static void employeeJob(int empNum,
                               java.lang.Integer[] jobCode)
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getJobcode =
        conn.prepareStatement("SELECT jobcode " +
                              "FROM samdbcat.persnl.employee " +
                              "WHERE empnum = ?");

    getJobcode.setInt(1, empNum);
    ResultSet rs = getJobcode.executeQuery();
    rs.next();
    int num = rs.getInt(1);
    if (rs.wasNull())
        jobCode[0] = null;
    else
        jobCode[0] = new Integer(num);
    rs.close();

    conn.close();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.persnl.employeejob(IN empnum INT,
                                             OUT jobcode INT)
  EXTERNAL NAME 'Payroll.employeeJob(int, java.lang.Integer[])'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the EMPLOYEEJOB procedure in MXCI:

```
CALL samdbcat.persnl.employeejob(337, ?);
```

The EMPLOYEEJOB procedure accepts the employee number 337 and returns this output in MXCI:

```
JOBCODE
-----------

        900

--- SQL operation complete.
```

The job code for employee number 337 is 900.

SUPPLIERINFO Stored Procedure

Java Method

```
public static void supplierInfo(BigDecimal suppNum,
                                String[] suppName,
                                String[] streetAddr,
                                String[] cityName,
                                String[] stateName,
                                String[] postCode)
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getSupplier =
        conn.prepareStatement("SELECT suppname, street, city, "+
                              "  state, postcode " +
                              "FROM samdbcat.invent.supplier " +
                              "WHERE suppnum = ?");

    getSupplier.setBigDecimal(1, suppNum);
    ResultSet rs = getSupplier.executeQuery();
    rs.next();
    suppName[0] = rs.getString(1);
    streetAddr[0] = rs.getString(2);
    cityName[0] = rs.getString(3);
    stateName[0] = rs.getString(4);
    postCode[0] = rs.getString(5);
    rs.close();

    conn.close();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.invent.supplierinfo(IN empnum NUMERIC(4),
                                              OUT suppname CHAR(18),
                                              OUT address CHAR(22),
                                              OUT city CHAR(14),
                                              OUT state CHAR(12),
                                              OUT zipcode CHAR(10))
  EXTERNAL NAME 'Inventory.supplierInfo'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the SUPPLIERINFO procedure in MXCI:

```
CALL samdbcat.invent.supplierinfo(25, ?, ?, ?, ?, ?);
```

The SUPPLIERINFO procedure accepts the supplier number 25 and returns this output in MXCI:

```
SUPPNAME            ADDRESS                 CITY            STATE        ZIPCODE
------------------  ----------------------  --------------  ------------ ----------

Schroeder's Ltd     212 Strasse Blvd West   Hamburg         RhodeIsland  22222

--- SQL operation complete.
```

Supplier number 25 is Schroeder's Ltd. and is located in Hamburg, Rhode Island.

SUPPLYNUMBERS Stored Procedure

Java Method

```
public static void supplyQuantities(int[] avgQty,
                                    int[] minQty,
                                    int[] maxQty)
    throws SQLException
{
    Connection conn = DriverManager.getConnection("jdbc:default:connection");

    PreparedStatement getQty =
        conn.prepareStatement("SELECT AVG(qty_on_hand), " +
                              "  MIN(qty_on_hand), " +
                              "  MAX(qty_on_hand) " +
                              "FROM samdbcat.invent.partloc");

    ResultSet rs = getQty.executeQuery();
    rs.next();
    avgQty[0] = rs.getInt(1);
    minQty[0] = rs.getInt(2);
    maxQty[0] = rs.getInt(3);
    rs.close();

    conn.close();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE samdbcat.invent.supplynumbers(OUT avrg INT,
                                               OUT minm INT,
                                               OUT maxm INT)
  EXTERNAL NAME 'Inventory.supplyQuantities'
  EXTERNAL PATH '/usr/mydir/myclasses'
  LANGUAGE JAVA
  PARAMETER STYLE JAVA
  READS SQL DATA;
```

CALL Statement to Invoke the SPJ

To invoke the SUPPLYNUMBERS procedure in MXCI:

```
CALL samdbcat.invent.supplynumbers(?, ?, ?);
```

The SUPPLYNUMBERS procedure returns this output in MXCI:

```
AVRG         MINM         MAXM
-----------  -----------  -----------

        167            0         1132

--- SQL operation complete.
```

The average number of items in inventory is 167, the minimum number is 0, and the maximum number is 1132.

ORDERSUMMARY Stored Procedure

Java Method

```
public static void orderSummary(java.lang.String onOrAfter,
                                long[] numOrders,
                                java.sql.ResultSet[] orders,
                                java.sql.ResultSet[] detail)
throws SQLException
{
java.lang.String s;
java.sql.Connection c =
DriverManager.getConnection("jdbc:default:connection");

// Get the number of orders on or after this date
s = " SELECT count(ordernum) FROM sales.orders " +
    " WHERE order_date >= cast(? as date) ";
java.sql.PreparedStatement ps1 = c.prepareStatement(s);
ps1.setString(1, onOrAfter);
java.sql.ResultSet rs = ps1.executeQuery();
rs.next();
numOrders[0] = rs.getLong(1);
rs.close();
// Open a result set for <order num, order info>
rows
s = " SELECT  AMOUNTS.*, ORDERS.order_date, EMPS.last_name           " +
    " FROM    ( select o.ordernum, count(d.partnum) as num_parts,    " +
    "             sum(d.unit_price * d.qty_ordered) as amount        " +
    "               from sales.orders o, sales.odetail d             " +
    "               where o.ordernum = d.ordernum                    " +
    "                 and o.order_date >= cast(? as date)            " +
    "               group by o.ordernum ) AMOUNTS,                   " +
    "              sales.orders ORDERS, persnl.employee EMPS         " +
    " WHERE       AMOUNTS.ordernum = ORDERS.ordernum                 " +
    "   AND       ORDERS.salesrep = EMPS.empnum                      " +
    " ORDER BY    ORDERS.ordernum                                    ";
java.sql.PreparedStatement ps2 = c.prepareStatement(s);
ps2.setString(1, onOrAfter);
orders[0] = ps2.executeQuery();
// Open a result set for order detail rows
s = " SELECT   D.*, P.partdesc                                       " +
    " FROM     sales.odetail D, sales.parts P, sales.orders O        " +
    " WHERE    D.partnum = P.partnum AND D.ordernum = O.ordernum     " +
    "   AND    O.order_date >= cast(? as date)                       " +
    " ORDER BY D.ordernum                                            ";
java.sql.PreparedStatement ps3 = c.prepareStatement(s);
ps3.setString(1, onOrAfter);
detail[0] = ps3.executeQuery();
}
```

CREATE PROCEDURE Statement

```
CREATE PROCEDURE SAMDBCAT.SALES.ORDER_SUMMARY
(IN ON_OR_AFTER_DATE VARCHAR(20) CHARACTER SET ISO88591,
OUT NUM_ORDERS
LARGEINT)
DYNAMIC RESULT SETS 2
EXTERNAL NAME 'SPJMethods.orderSummary(java.lang.String,
long[],java.sql.ResultSet[], java.sql.ResultSet[])'
EXTERNAL PATH '/usr/mydir/myclasses';
LOCATION \ALPINE.$SYSTEM.ZSDCR2C6.L1Z7NW00
LANGUAGE JAVA
PARAMETER STYLE JAVA
READS SQL DATA
NOT DETERMINISTIC
```

```
ISOLATE
;
--- SQL operation complete.
```

CALL Statement to Invoke the SPJ

```
CALL samdbcat.sales.order_summary('2001-01-01', ?);
```

The ORDER_SUMMARY procedure accepts the date and returns this output in MXCI:

```
NUM_ORDERS
--------------------
                  13
ORDERNUM   NUM_PARTS    AMOUNT           Order/Date  Last Name
---------- -------------- --------------- ---------- ------------------
    100210           4       19020.00    2003-04-10 HUGHES
    100250           4       22625.00    2003-01-23 HUGHES
    101220           4       45525.00    2003-07-21 SCHNABL
    200300           3       52000.00    2003-02-06 SCHAEFFER
    200320           4        9195.00    2003-02-17 KARAJAN
    200490           2        1065.00    2003-03-19 WEIGL
      .
      .
      .
--- 13 row(s) selected.

Order/Num  Part/Num Unit/Price   Qty/Ord    Part Description
---------- -------- ------------ ---------- ------------------
    100210     2001     1100.00           3 GRAPHIC PRINTER,M1
    100210     2403      620.00           6 DAISY PRINTER,T2
    100210      244     3500.00           3 PC GOLD, 30 MB
    100210     5100      150.00          10 MONITOR BW, TYPE 1
    100250     6500       95.00          10 DISK CONTROLLER
    100250     6301      245.00          15 GRAPHIC CARD, HR
      .
      .
      .
--- 70 row(s) selected
--- SQL operation complete.
```

Index