Java in Adaptive Server® Enterprise

Adaptive Server® Enterprise

Version 12

Document ID: 31652-01-1200-01

Last revised: October 1999

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

Contents

About This Book

This book describes how to install and use Java classes and methods in the Sybase® Adaptive Server® Enterprise database.

Audience

This book is for Sybase System Administrators, Database Owners, and users who are familiar with the Java programming language and Transact-SQL®, the Sybase version of Structured Query Language (SQL).

How to use this book

This book will assist you in installing, configuring, and using Java classes and methods in the Adaptive Server database. It includes these chapters:

- Chapter 1, “An Introduction to Java in the Database” provides an overview of Java in Adaptive Server, including a Questions and Answers section for both novice and experienced Java users.
- Chapter 2, “Preparing for and Maintaining Java in the Database” describes the Java runtime environment and the steps for enabling Java on the server and installing Java classes.
- Chapter 3, “Using Java Classes in SQL” describes how to use Java-SQL in your Adaptive Server database.
- Chapter 4, “Data Access Using JDBC” describes how you use a JDBC driver (on the server or on the client) to perform SQL operations in Java.
- Chapter 5, “XML in the Database” describes how you can use Java to access Extensible Markup Language (XML) documents from an Adaptive Server database.
- Chapter 6, “Debugging Java in the Database” describes how you use the Sybase debugger with Java.
- Chapter 7, “Reference Topics” provides information about datatype mapping, Java-SQL syntax, and other useful information.

In addition, a glossary provides descriptions of Java and Java-SQL terms used in this book.

Related documents

The following documents comprise the Sybase Adaptive Server Enterprise documentation:

- The *Release Bulletin* for your platform – contains last-minute information that was too late to be included in the books.

  A more recent version of the *Release Bulletin* may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use SyBooks™-on-the-Web.

- The Adaptive Server installation documentation for your platform – describes installation and upgrade procedures for all Adaptive Server and related Sybase products.

- *What's New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server release 12.0, the system changes added to support those features, and the changes that may affect your existing applications.

- *Transact-SQL User's Guide* – documents Transact-SQL", Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the *pubs2* and *pubs3* sample databases.

- *System Administration Guide* – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.

- *Adaptive Server Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and datatypes. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.

- *Performance and Tuning Guide* – explains how to tune Adaptive Server for maximum performance. This manual includes information about database design issues that affect performance, query optimization, how to tune Adaptive Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.

- The *Utility Programs* manual for your platform – documents the Adaptive Server utility programs, such as **isql** and **bcp**, which are executed at the operating system level.

- *Error Messages* and *Troubleshooting Guide* – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.

- *Component Integration Services User's Guide for Adaptive Server Enterprise and OmniConnect* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase's Failover to configuring an Adaptive Server as a companion server in a high availability system.
- *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM Features in distributed transaction processing environments.
- *Adaptive Server Glossary* – defines technical terms used in the Adaptive Server documentation.

Other sources of information

Use the Sybase Technical Library CD and the Technical Library Product Manuals web site to learn more about your product:

- Technical Library CD contains product manuals and technical documents and is included with your software. The DynaText browser (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

  Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting Technical Library.
- Technical Library Product Manuals web site is an HTML version of the Technical Library CD that you can access using a standard web browser. In addition to product manuals, you'll find links to the Technical Documents web site (formerly known as Tech Info Library), the Solved Cases page, and Sybase/Powersoft newsgroups.

  To access the Technical Library Product Manuals web site, go to Product Manuals at http://sybooks.sybase.com.

Sybase certifications on the web

Technical documentation at the Sybase web site is updated frequently.

❖ **For the latest information on product certifications and/or the EBF Rollups:**

1. Point your web browser to Technical Documents at http://techinfo.sybase.com.
2. In the Browse section, click on What's Hot.
3. Select links to Certification Reports and EBF Rollups, as well as links to Technical Newsletters, online manuals, and so on.

❖ **If you are a registered SupportPlus user:**

1 Point your web browser to Technical Documents at http://techinfo.sybase.com.

2 In the Browse section, click on What's Hot.

3 Click on EBF Rollups.

You can research EBFs using Technical Documents, and you can download EBFs using Electronic Software Distribution (ESD).

4 Follow the instructions associated with the SupportPlus^{SM} Online Services entries.

❖ **If you are not a registered SupportPlus user, and you want to become one:**

You can register by following the instructions on the Web.

To use SupportPlus, you need:

1 A Web browser that supports the Secure Sockets Layer (SSL), such as Netscape Navigator 1.2 or later

2 An active support license

3 A named technical support contact

4 Your user ID and password

❖ **Whether or not you are a registered SupportPlus user:**

You may use Sybase's Technical Documents. Certification Reports are among the features documented at this site.

1 Point your web browser to Technical Documents at http://techinfo.sybase.com

2 In the Browse section, click on What's Hot.

3 Click on the topic that interests you.

Java syntax conventions

This book uses these font and syntax conventions for Java items:

- Classes, interfaces, methods, and packages are shown in bold Helvetica within paragraph text. For example:

  SybConnection class

  SybEventHandler interface

  setBinaryStream( ) method

com.Sybase.jdbx package

- Objects, instances, and parameter names are shown in italics. For example:

  "In the following example, *ctx* is a **DirContext** object."

  "*eventHdler* is an instance of the **SybEventHandler** class that you implement."

  "The *classes* parameter is a string that lists specific classes you want to debug."

- Java names are always case sensitive. For example, if a Java method name is shown as **Misc.stripLeadingBlanks( )**, you must type the method name exactly as displayed.

Transact-SQL syntax conventions

This book uses the same font and syntax conventions for Transact-SQL as other Adaptive Server documents:

- Command names, command option names, utility names, utility flags, and other keywords are in bold Helvetica in paragraph text. For example:

  select command

  isql utility

  -f flag

- Variables, or words that stand for values that you fill in, are in italics. For example:

  user_name

  server_name

- Code fragments are shown in a monospace font.Variables in code fragments (that is, words that stand for values that you fill in) are italicized. For example:

  ```
  Connection con = DriverManager.getConnection
  ("jdbc:sybase:Tds:host:port", props);
  ```

- You can disregard case when typing Transact-SQL keywords. For example, **SELECT**, **Select**, and **select** are the same.

Additional conventions for syntax statements in this manual are described in Table 1. Examples illustrating each convention can be found in the *System Administration Guide*.

Table 1: Syntax statement conventions

Key	Definition
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.
[ ]	Brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you may select only one of the options shown.
,	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

An Introduction to Java in the Database

This chapter provides an overview of Java classes in Adaptive Server Enterprise.

These topics are discussed:

Advantages of Java in the Database

Adaptive Server provides a runtime environment for Java, which means that Java code can be executed in the server. Building a runtime environment for Java in the database server provides powerful new ways of managing and storing both data and logic.

- You can use the Java programming language as an integral part of Transact-SQL.
- You can reuse Java code in the different layers of your application—client, middle-tier, or server—and use them wherever makes most sense to you.
- Java in Adaptive Server provides a more powerful language than stored procedures for building logic into the database.
- Java classes become rich, user-defined data types.
- Methods of Java classes provide new functions accessible from SQL.
- Java can be used in the database without jeopardizing the integrity, security, and robustness of the database. Using Java does not alter the behavior of existing SQL statements or other aspects of non-Java relational database behavior.

Capabilities of Java in the Database

Java in Adaptive Server provides these functionalities:

- Java user-defined functions (UDFs)
- Java classes as datatypes in SQL

Java User-Defined Functions

You can install Java classes in the Adaptive Server database, and then invoke the methods of those classes, both from within the SQL system and from client systems.

The methods of an object-oriented language correspond to the functions of a procedural language. You can invoke Java methods as UDFs in, for example, **select** lists and **where** clauses. You can use methods from other sources or methods you create and test.

Java Classes as Datatypes

With Java in the database, you can install pure Java classes in a SQL system, and then use those classes in a natural manner as datatypes in SQL. This capability adds a full object-oriented datatype extension mechanism to SQL, using a model that is widely understood and a language that is portable and widely available. The objects that you create and store with this facility are readily transferable to any Java-enabled environment, either in another SQL system or stand-alone Java environment.

This capability of using Java classes in the database has two different but complementary uses:

- It provides a type extension mechanism for SQL, which you can use for data that is created and processed in SQL.
- It provides a persistent data capability for Java, which you can use to store data in SQL that is created and processed (mainly) in Java. Java in Adaptive Server provides a distinct advantage over traditional SQL facilities: You do not need to map the Java objects into scalar SQL data types or store the Java objects as untyped binary strings.

Standards

The SQLJ consortium of SQL vendors is developing specifications for using Java with SQL. The consortium submits these specifications to ANSI for formal processing as standards.

The SQLJ specifications are divided into three parts:

- *Part 0* – specifications for embedding SQL statements in Java methods, similar to the traditional SQL facilities for embedded SQL in COBOL and C and other languages. The Java classes containing embedded SQL statements are precompiled to pure Java classes with JDBC calls.
- *Part 1* – specifications for installing Java classes in a SQL system, and for invoking Java static methods as SQL stored procedures and functions.
- *Part 2* – specifications for using Java classes as SQL datatypes.

You can use methods and classes using the Part 0 specifications with Java in Adaptive Server.

Java in Adaptive Server provided the basis for Parts 1 and 2. However, Java in Adaptive Server allows you to use Java names directly in SQL, whereas SQLJ Parts 1 and 2 currently require that you use the SQL **create** statement to define SQL aliases for Java method and class names. Java in Adaptive Server will support the SQLJ Parts 1 and 2 specifications when they are finalized.

Java in the Database: Questions and Answers

Although this book assumes that readers are familiar with Java, there is much to learn about Java in a database. Sybase is not only extending the capabilities of the database with Java, but also extending the capabilities of Java with the database.

Both experienced and novice Java users should read this section. It uses a question-and-answer format to familiarize you with the basics of Java in Adaptive Server.

What Are the Key Features?

All of these points are explained in detail in later sections. With Java in Adaptive Server, you can:

- Run Java in the database server using an internal Java Virtual Machine (Java VM).
- Call Java functions (methods) from SQL statements.
- Access data from Java using an internal JDBC driver.
- Use Java classes as datatypes.
- Save instances of Java objects in tables.
- Generate XML-formatted documents from data stored in Adaptive Server databases and, conversely, store XML documents and data extracted from them in Adaptive Server databases.
- Debug Java in the database.
- Preserve the behavior of existing SQL statements and other aspects of non-Java relational database behavior.

How Can I Store Java Instructions in the Database?

Java is an object-oriented language, so its instructions (source code) come in the form of classes. You write and compile the Java instructions outside the database into compiled classes (byte code), which are binary files holding Java instructions.

You then install the compiled classes into the database, where they can be executed in the database server.

Adaptive Server is a runtime environment for Java classes. You need a Java development environment, such as Sybase PowerJ™ or Sun Microsystems Java Development Kit (JDK), to write and compile Java.

How Is Java Executed in the Database?

To support Java in the database, Adaptive Server:

- Comes with its own Java VM, specifically developed for handling Java processing in the server.
- Uses its own JDBC driver that runs in the server and accesses a database.

The Sybase Java VM runs in the database environment. It interprets compiled Java instructions and runs them in the database server.

The Sybase Java VM meets the JCM specifications from JavaSoft; it is designed to work with the 1.1.6 version of the Java API. It supports public class and instance methods; classes inheriting from other classes; the Java API; and access to **protected**, **public**, and **private** fields. Some Java API functions not appropriate in a server environment, such as user interface elements, are not supported. All supported Java API packages and classes come with Adaptive Server.

The Adaptive Server Java VM is available at all times to perform a Java operation whenever it is required as part of the execution of a SQL statement. The database server starts the Java VM automatically when it is needed; you do not need to take any explicit action to start or stop the Java VM.

Client- and Server-Side JDBC

JDBC is the industry standard API for executing SQL in Java.

Adaptive Server provides its own server-side JDBC driver. This driver is designed to maximize performance as it executes on the server because it does not need to communicate across the network. This internal driver permits Java classes installed in a database to use JDBC classes that execute SQL statements.

When JDBC classes are used within a client application, you typically must use jConnect® for JDBC™, the Sybase client-side JDBC database driver, to provide the classes necessary to establish a database connection.

How Can I Use Java and SQL Together?

A guiding principle for the design of Java in the database is that it provides a natural, open extension to existing SQL functionality.

- *Java operations are invoked from SQL*– Sybase has extended the range of SQL expressions to include fields and methods of Java objects, so that Java operations can be included in a SQL statement.
- *Java classes become user-defined datatypes* – You store Java classes using the same SQL statements as those used for traditional SQL datatypes.

You can use classes that are part of the Java API and classes created and compiled by Java developers. The Java API classes are created and compiled by Sun Microsystems and by Sybase.

What Is the Java API?

The Java Application Programmer's Interface (API) is a set of classes defined by Sun Microsystems. It provides a range of base functionality that can be used and extended by Java developers. It is the core of "what you can do" with Java.

The Java API offers considerable functionality in its own right. A large portion of the Java API is built in to any database that is enabled to use Java code—which includes the majority of non-visual classes from the Java API already familiar to developers using the Sun Microsystems JDK.

You can use the Java API in classes, in stored procedures, and in SQL statements. You can treat the Java API classes as extensions to the available built-in functions provided by SQL.

How Can I Access the Java API from SQL?

You can use the Java API in classes, in stored procedures, and in SQL statements. You can create the Java API classes as extensions to the available built-in functions provided by SQL.

For example, the SQL function PI(*) returns the value for Pi. The Java API class **java.lang.Math** has a parallel field named *PI* that returns the same value. But **java.lang.Math** also has a field named *E* that returns the base of the natural logarithm, as well as a method that computes the remainder operation on two arguments as prescribed by the IEE754 standard.

Which Java Classes Are Supported in the Java API?

Not all Java API classes are supported in the database. Some classes, for example the **java.awt** package that contains user interface components for applications, is not appropriate inside a database server. Other classes, including part of **java.io**, deal with writing information to a disk, and this also is not supported in the database server environment.

Can I Install My Own Java Classes?

You can install your own Java classes into the database as, for example, a user-created **Employee** class or **Inventory** class that a developer designed, wrote, and compiled with a Java compiler.

User-defined Java classes can contain both information and methods. Once installed in a database, Adaptive Server lets you use these classes in all parts and operations of the database and execute their functionality (in the form of class or instance methods) as easily as calling a stored procedure.

Can I Access Data Using Java?

The JDBC interface is an industry standard designed to access database systems. The JDBC classes are designed to connect to a database, request data using SQL statements, and return results that can be processed in the client application.

You can connect from a client application to Adaptive Server Enterprise via JDBC, using jConnect or a JDBC/ODBC bridge. Adaptive Server also provides an internal JDBC driver, which permits Java classes installed in a database to use JDBC classes that execute SQL statements.

Can I Move Classes from Client to Server?

The Java in Adaptive Server design allows you to create Java classes that can be moved between levels of an enterprise application: The same Java class can be integrated into either the client application, a middle tier, or the database.

How Do I Use Java Classes in SQL?

Using Java classes, whether user-defined or from the Java API, in SQL is a three-step activity:

1 Write or acquire a set of Java classes that you want to use as SQL datatypes.

2 Install those classes in the Adaptive Server database.

3 Use those classes in SQL code:

- Call class (static) methods of those classes as UDFs.
- Declare the Java classes as datatypes of SQL columns, variables, and parameters. In this book, they are called Java-SQL columns, variables, and parameters.
- Reference the Java-SQL columns, their fields, and their methods.

Can I Find More Information About Java in the Database?

There are many books about Java and Java in the database. Two particularly useful books are:

- James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley, 1996.
- Graham Hamilton and Rick Cattell, *JDBC: A Java SQL API*, Version 1.20, JavaSoft, January 10, 1997.

What You Cannot Do with Java in the Database

Adaptive Server is a runtime environment for Java classes, not a Java development environment.

You cannot carry out these tasks in the database:

- Edit class source files (**.java* files).
- Compile Java class source files (**.java* files).
- Execute Java APIs that are not supported, such as applet and visual classes.

In this release of Adaptive Server, certain other restrictions apply:

- If a Java method accesses the database through JDBC, result-set values are available *only* to the Java method, not to the client application.
- Output parameters are not supported. A method can manipulate the data it receives from a JDBC connection, but the only value it can return to its caller is a single return value declared as part of its definition.

Sample Java Classes

The chapters of this book use simple Java classes to illustrate basic principles for using Java in the database. You can find copies of these classes in the chapters that describe them and in the Sybase release directory in *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE%\Ase-12_0\sample\JavaSql* (Windows NT). This subdirectory also contains Javadoc facilities so that you can view specifications about sample classes and methods with your Web browser.

CHAPTER 2

Preparing for and Maintaining Java in the Database

This chapter describes the Java runtime environment, how to enable Java on the server, and how to install and maintain Java classes in the database.

These topics are discussed:

The Java Runtime Environment

The Adaptive Server runtime environment for Java requires a Java VM, which is available as part of the database server, and the Sybase runtime Java classes, or Java API. If you are running Java applications on the client, you may also require the Sybase JDBC driver, jConnect, on the client.

Java Classes in the Database

You can use either of the following sources for Java classes:

- Sybase runtime Java classes
- User-defined classes

Sybase Runtime Java Classes

The Sybase Java VM supports a subset of JDK version 1.1.6 (UNIX and Windows NT) classes and packages.

The Sybase runtime Java classes are the low-level classes installed to Java-enable a database. They are downloaded when Adaptive Server is installed and are available thereafter from *$SYBASE /$SYBASE_ASE/lib/runtime.zip* (UNIX) or *%SYBASE%\%SYBASE_ASE%\lib\runtime.zip* (Windows NT). You do not need to set the CLASSPATH environment variable specifically for Java in Adaptive Server.

Sybase does not support runtime Java packages and classes that assume a screen display, deal with networking and remote communications, or handle security. See Chapter 7, “Reference Topics” for a list of supported and not-supported packages and classes.

User-Defined Java Classes

You install user-defined classes into the database using the **installjava** utility. Once installed, these classes are available from other classes in the database and from SQL as user-defined datatypes.

JDBC Drivers

The Sybase internal JDBC driver that comes with Adaptive Server supports JDBC version 1.1.

If your system requires a JDBC driver on the client, you must use jConnect version 4.1, which also supports JDBC version 1.1.

Enabling the Server for Java

To enable the server and its databases for Java, enter this command from **isql**:

```
sp_configure “enable java”, 1
```

Then shutdown and reboot the server.

By default, Adaptive Server is not enabled for Java. You cannot install Java classes or perform any Java operations until the server is enabled for Java.

You can increase or decrease the amount of memory available for Java in Adaptive Server and optimize performance using the **sp_configure** system procedure. Java configuration parameters are described in the *System Administration Guide*.

Disabling the Server for Java

To disable Java in the database, enter this command from **isql**:

```
sp_configure “enable java”, 0
```

Creating Java Classes and JARs

The Sybase-supported classes from the JDK are installed on your system when you install Adaptive Server version 12 or later. This section describes the steps for creating and installing your own Java classes.

To make your Java classes (or classes from other sources) available for use in the server, follow these steps:

1 Write and save the Java code that defines the classes.

2 Compile the Java code.

3 Create Java archive (JAR) files to organize and contain your classes.

4 Install the JARs/classes in the database.

Writing the Java Code

Use the Sun Java SDK or a development tool such as Sybase PowerJ to write the Java code for your class declarations. Save the Java code in a file with an extension of *.java*. The name and case of the file must be the same as that of the class.

Note Make certain that any Java API classes used by your classes are among the supported API classes listed in Chapter 7, “Reference Topics”.

Compiling Java Code

This step turns the class declaration containing Java code into a new, separate file containing byte code. The name of the new file is the same as the Java code file but has an extension of *.class*. You can run a compiled Java class in a Java runtime environment regardless of the platform on which it was compiled or the operating system on which it runs.

Saving Classes in a JAR File

You can organize your Java classes by collecting related classes in packages and storing them in JAR files.

To install Java classes in a database, the classes or packages *must* first be saved in a JAR file, in uncompressed form. To create an uncompressed JAR file that contains Java classes, use the Java **jar cf0** command.

In this UNIX example, the **jar** command creates an uncompressed JAR file that contains all *.class* files in the **jcsPackage** directory:

```
jar cf0 jcsPackage.jar jcsPackage/*.class
```

Note that the "0" in **cf0** is "zero."

JAR files allow you to install or remove related classes as a group.

Installing Java Classes in the Database

To install Java classes from a client operating system file, use the **installjava** (UNIX) or **instjava** (Windows NT) utility from the command line.

Refer to *Adaptive Server Utilities Programs* for your platform for detailed information about these utilities. Both utilities perform the same tasks; for simplicity, this document uses UNIX examples.

Using *installjava*

installjava copies a JAR file into the Adaptive Server system and makes the Java classes contained in the JAR available for use in the current database. The syntax is:

```
installjava
 -f file_name
 [-new | -update]
 [-j jar_name]
 …
```

For example, to install classes in the *addr.jar* file, enter:

```
installjava -f "/home/usera/jars/addr.jar"
```

The **–f** parameter specifies an operating system file that contains a JAR. You must use the complete path name for the JAR.

This section describes retained JAR files (using **-j)** and updating installed JARs and classes (using **new** and **update**). For more information about these and the other options available with **installjava**, see the *Utility Programs* manual for your platform.

Retaining the JAR File

When a JAR is installed in a database, the server disassembles the JAR, extracts the classes, and stores them separately. The JAR is not stored in the database unless you specify **installjava** with the **-j** parameter.

Use of **-j** determines whether the Adaptive Server system retains the JAR specified in **installjava** or uses the JAR only to extract the classes to be installed.

- If you do not specify the **-j** parameter, the Adaptive Server system does not retain any association of the classes with the JAR. This is the default option.

- If you do specify the **-j** parameter, Adaptive Server installs the classes contained in the JAR in the normal manner, and then retains the JAR and its association with the installed classes.

If you retain the JAR file:

- You can remove the JAR and all classes associated with it, all at once, with the **remove java** statement. Otherwise, you must remove each class or package of classes one at a time.
- Other systems may request that the class associated with a given Java column be downloaded with the column value. If a class retains its association with the JAR, the Adaptive Server system can download the JAR, rather than individual classes.

Updating Installed Classes

The **new** and **update** clauses of **installjava** indicate whether you want new classes to replace currently installed classes.

- If you specify **new**, you cannot install a class with the same name as an existing class.
- If you specify **update**, you can install a class with the same name as an existing class, and the newly installed class replaces the existing class.

Warning! If you alter a class used as a column datatype by reinstalling a modified version of the class, make sure that the modified class can read and use existing objects (rows) in tables using that class as a datatype. Otherwise, you may be unable to access existing objects without reinstalling the class.

Substitution of new classes for installed classes depends also on whether the classes being installed or the already installed classes are associated with a JAR. Thus:

- If you update a JAR, all classes in the existing JAR are deleted and replaced with classes in the new JAR.
- A class can only be associated with a single JAR. You cannot install a class in one JAR if a class of that same name is already installed and associated with another JAR. Similarly, you cannot install a class not-associated with a JAR if that class is currently installed and associated with a JAR.

You can, however, install a class in a retained JAR with the same name as an installed class not associated with a JAR. In this case, the class not associated with a JAR is deleted and the new class of the same name is associated with the new JAR.

If you want to reorganize your installed classes in new JARs, you may find it easier to first disassociate the affected classes from their JARs. See "Retaining Classes" on page 24 for information about disassociating classes from JARs.

Referencing Other Java-SQL Classes

Installed classes can reference other classes in the same JAR file and classes previously installed in the same database, but they cannot references classes in other databases.

If the classes in a JAR file do reference undefined classes, an error may result:

- If an undefined class is referenced directly in SQL, it causes a syntax error for "undefined class."
- If an undefined class is referenced within a Java method that has been invoked, it throws a Java exception that may be caught in the invoked Java method or cause the general SQL exception described in "Exceptions in Java-SQL Methods" on page 35.

The definition of a class can contain references to unsupported classes and methods as long as they are not actively referenced or invoked. Similarly, an installed class can contain a reference to a user-defined class that is not installed in the same database as long as the class is not instantiated or referenced.

Viewing Information about Installed Classes and JARs

To view information about classes and JARs installed in the database, use the **sp_helpjava** system procedure.The syntax is:

sp_helpjava ['class' [, *name* [,detail]] | 'jar' [, *name*]]

To view detailed information about the **Address** class, for example, log in to **isql** and enter:

```
sp_helpjava "class", Address, detail
```

Refer to "sp_helpjava" in the *Reference Manual* for more information.

Downloading Installed Classes and JARs

You can download copies of Java classes installed on one database for use in other databases or applications.

Use the **extractjava** system utility to download a JAR file and its classes to a client operating system file. For example, to download *addr.jar* to *~/home/usera/jars/addrcopy.jar*, enter:

```
extractjava –j ‘addr.jar‘  -f
 ‘~/home/usera/jars/addrcopy.jar'
```

Refer to the *Adaptive Server Utility Programs* manual for more information about this utility.

Removing Classes and JARs

Use the Transact-SQL **remove java** statement to uninstall one or more Java-SQL classes from the database. **remove java** can specify one or more Java class names, Java package names, or retained JAR names. For example, to uninstall the package **utilityClasses**, from **isql** enter:

```
remove java package "utilityClasses"
```

Note Adaptive Server does not allow you to remove classes that are used as the datatypes for columns and parameters. Make sure that you do not remove subclasses or classes that are used as variables or UDF return types.

When you specify **remove java package** the command deletes all classes in the specified package and all of its sub-packages.

See the *Reference Manual* for more information about **remove java**.

Retaining Classes

You can delete a JAR file from the database but retain its classes as classes no longer associated with a JAR. Use **remove java** with the **retain classes** option if, for example, you want to rearrange the contents of several retained JARs.

For example, from **isql** enter:

```
remove java jar 'utilityClasses' retain classes
```

Once the classes are disassociated from their JARs, you can associate them with new JARs using **installjava update**.

CHAPTER 3

Using Java Classes in SQL

This chapter describes how to use Java classes in an Adaptive Server environment. The first sections give you enough information to get started; succeeding sections provide more advanced information.

These topics are discussed:

In this document, SQL columns and variables whose datatypes are Java-SQL classes are described as Java-SQL columns and Java-SQL variables or as Java-SQL data items.

The sample classes used in this chapter can be found in “Sample Java Classes” on page 11 and in *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE%\Ase-12_0\sample\JavaSql* (Windows NT).

General Considerations

Before you use Java in your Adaptive Server database, here are some general considerations.

- Java-SQL classes contain:
  - Fields that have declared Java datatypes
  - Methods whose parameters and results have declared Java datatypes
  - Java datatypes for which there are corresponding SQL datatypes are defined in “Datatype Mapping Between Java and SQL” on page 166.
- Java-SQL classes can include classes, fields, and methods that are **private**, **protected**, **friendly**, or **public**.

  Classes, fields and methods that are **public** can be referenced in SQL. Classes, fields, and methods that are **private**, **protected**, or **friendly** cannot be referenced in SQL, but they can be referenced in Java, and are subject to normal Java rules.
- Java-SQL classes, fields, and methods all have various syntactic properties:
  - Classes – the number of fields and their names
  - Field – their datatypes
  - Methods – the number of parameters and their datatypes, and the datatype of the result

  The SQL system determines these syntactic properties from the Java-SQL classes themselves, using the Java Reflection API.

Java-SQL Names

Java-SQL class names (identifiers) are limited to 255 bytes. Java-SQL field and method names can be any length, but they must be 255 bytes or less if you use them in Transact-SQL. All Java-SQL names must conform to the rules for Transact-SQL identifiers if you use them in Transact-SQL statements.

Class, field, and method names of 30 or more bytes must be surrounded by quotation marks.

The first character of the name must be either an alphabetic character (uppercase or lowercase) or an underscore (_) symbol. Subsequent characters can include alphabetic characters, numbers, the dollar ($) symbol, or the underscore (_) symbol.

Java-SQL names are always case sensitive, regardless of whether the SQL system is specified as case sensitive or case insensitive.

See **Java-SQL Identifiers on page 168** for more information about identifiers.

Using Java Classes as Datatypes

After you have installed a set of Java classes, you can reference them as datatypes in SQL. To be used as a column datatype, a Java-SQL class must be defined as **public** and must implement either **java.io.Serializable** or **java.io.Externalizable**.

You can specify Java-SQL classes as:

- The datatypes of SQL columns
- The datatypes of Transact-SQL variables and parameters to Transact-SQL stored procedures
- Default values for SQL columns

When you create a table, you can specify Java-SQL classes as the datatypes of SQL columns:

```
create table emps (
   name varchar(30),
   home_addr Address,
   mailing_addr Address2Line null )
```

The *name* column is an ordinary SQL character string, the *home_addr* and *mailing_addr* columns can contain Java objects, and **Address** and **Address2Line** are Java-SQL classes that have been installed in the database.

You can specify Java-SQL classes as the datatypes of Transact-SQL variables:

```
declare @A Address
declare @A2 Address2Line
```

You can also specify default values for Java-SQL columns, subject to the normal constraint that the specified default must be a constant expression. This expression is normally a constructor invocation using the **new** operator with constant arguments, such as the following:

```
create table emps (
   name varchar(30),
   home_addr Address default new Address
      ('Not known', ''),
   mailing_addr Address2Line
)
```

Creating Tables with Java-SQL Columns

When you create or alter tables with Java-SQL columns, you can specify any installed Java class as a column datatype. You can also specify how the information in the column is to be stored. Your choice of storage options influences the speed of referencing and updating the specified fields and whether they can be indexed.

Column values for a row normally are stored "in row," that is, consecutively on the data pages allocated to a table. However, you can choose to store Java-SQL columns in a separate "off row" location in the same way that text and image data items are stored. The default for Java-SQL columns is off row.

If a Java-SQL column is stored in row:

- Java objects are processed faster than objects that are stored off row.
- An object stored in row cannot occupy more than 255 bytes. This includes its entire serialization, not just the values in its fields. A Java object whose runtime representation is more than 255 bytes generates an exception, and the command aborts.

Note You can use the **datalength** system function to find the length of the object. See the *Reference Manual* for information about **datalength**.

If a Java-SQL column is stored off row, the column is subject to the restrictions that apply to text and image columns:

- The column cannot be referenced in a check constraint.
- The column cannot be included in the column select list of a select query with **select distinct**.
- The column cannot be specified in a comparison operator, in a predicate, or in a **group by** clause.

The syntax for **create table** with the **in row/off row** option is:

```
create table...column_name datatype
    [default {constant_expression | user | null}]
  {[{identity | null | not null}]
    [off row | in row]...
```

Similarly, the syntax for **alter table** is:

```
alter table...{add column_name datatype
  [default {constant_expression | user | null}]
  {identity | null} [off row | in row]...
```

The following code fragment alters the *emps* table, adding a new column *vacation_addr* with an **Address** datatype:

```
alter table emps add vacation_addr Address null
```

Selecting, Inserting, Updating, and Deleting Java Objects

After you specify Java-SQL columns, the values that you assign to those data items must be Java instances. Such instances are generated initially by calls to Java constructors using the **new** operator. You can generate Java instances for both columns and variables.

A constructor method has the same name as the class, and has no declared datatype. If you do not include a constructor method in your class definition, a default method is provided by the Java base object. You can supply more than one constructor for each class, with different numbers and types of arguments. When a constructor is invoked, the one with the proper number and type of arguments is used.

In the following example, Java instances are generated for both columns and variables:

```
declare @A Address, @AA Address, @A2 Address2Line,
   @AA2 Address2Line

 select @A = new Address( )
 select @AA = new Address('123 Main Street', '99123')
 select @A2 = new Address2Line( )
 select @AA2 = new Address2Line('987 Front Street',
   'Unit 2', '99543')

 insert into emps values('John Doe', new Address( ),
   new Address2Line( ))
 insert into emps values('Bob Smith', new Address('432 Elm
   Street', '99654'), new Address2Line('PO Box 99',
   'attn: Bob Smith', '99678') )
```

Values assigned to Java-SQL columns and variables can then be assigned to other Java-SQL columns and variables. For example:

```
declare @A Address, @AA Address, @A2 Address2Line,
   @AA2 Address2Line

 select @A = home_addr, @A2 = mailing_addr from emps
   where name = 'John Doe'
 insert into emps values ('George Baker', @A, @A2)

 select @AA2 = @A2
 update emps
   set home_addr = new Address('456 Shoreline Drive', '99321'),
         mailing_addr =  @AA2
```

```
where name = 'Bob Smith'
```

You can also copy values of Java-SQL columns from one table to another. For example:

```
create table trainees (
   name char(30),
   home_addr Address,
   mailing_addr Address2Line null
 )
insert into trainees
select * from emps
   where name in ('Don Green', 'Bob Smith',
   'George Baker')
```

Referencing Java Fields in SQL

You can reference and update the fields of Java-SQL columns and of Java-SQL variables with normal SQL qualification. To avoid ambiguities with the SQL use of dots to qualify names, use a double-angle (>>) to qualify Java field and method names when referencing them in SQL.

```
declare @name varchar(100), @street varchar(100),
   @streetLine2 varchar(100), @zip char(10), @A Address

 select @A = new Address()
 select @A>>street = '789 Oak Lane'
 select @street = @A>>street

 select @street = home_addr>>street, @zip = home_addr>>zip from emps
   where name = 'Bob Smith'
 select @name = name from emps
   where home_addr>>street= '456 Shoreline Drive'

 update emps
   set home_addr>>street = '457 Shoreline Drive',
      home_addr>>zip = '99323'
   where home_addr>>street = '456 Shoreline Drive'
```

Invoking Java Methods in SQL

You can invoke Java methods in SQL by referencing them, with name qualification, on instances for instance methods, and on either instances or classes for class methods.

Instance methods are generally closely tied to the data encapsulated in an instance of their class. A class method is the same as a static method. Class methods often apply to objects and values from a wide range of classes.

Once you have installed a class method, it is ready for use. A class that contains a class method for use as a function must be **public**, but it does not need to be serializable.

One of the primary benefits of using Java with Adaptive Server is that you can use class methods that return a value to the caller as user-defined functions (UDFs).

You can use a Java class method as a UDF in a stored procedure, a trigger, a **where** clause, or anywhere that you can use a built-in SQL function.

Sample Methods

The sample **Address** and **Address2Line** classes have instance methods named **toString( )**, and the sample **Misc** class has class methods named **stripLeadingBlanks( )**, **getNumber( )**, and **getStreet( )**. You can invoke value methods as functions in a value expression.

```
declare @name varchar(100)
declare @street varchar(100)
declare @streetnum int
declare @A2 Address2Line

select @name = Misc.stripLeadingBlanks(name),
       @street = Misc.stripLeadingBlanks(home_addr>>street),
       @streetnum = Misc.getNumber(home_addr>>street),
       @A2 = mailing_addr
   from emps
   where home_addr>>toString( ) like '%Shoreline%'
```

For information about void methods (methods with no returned value) see “Type and Void Methods” on page 49.

Exceptions in Java-SQL Methods

When the invocation of a Java-SQL method completes with unhandled exceptions, a SQL exception is raised, and this error message displays:

```
Java method terminated with exception
```

The message text for the exception consists of the name of the Java class that raised the exception, followed by the character string (if any) supplied when the Java exception was thrown.

Representing Java Instances

When you select a Java-SQL data item in **isql**, Adaptive Server returns the object (the reference to the Java instance), not the value. Adaptive Server must use the object to interact with the Java VM.

If, for example, you want to use an actual string value, you must invoke a method that translates the object into a char or varchar value. The **toString( )** method in the **Address** class is an example of such a method. Use **toString( )** or a similar method for numbers when you want to:

- Display or print the value
- Use a datatype that does not map to a SQL datatype
- Compare values

When you use the **toString( )** method, Adaptive Server imposes a limit of 255 characters. The display software on your computer may truncate the data item further so that it fits on the screen without wrapping.

If you include a **toString( )** or similar method in each class, you can return the value of the object's **toString( )** method in either of two ways:

- You can select a particular field in the Java-SQL column, which automatically invokes **toString( )**:

  ```
  select home_addr>>street from emps
  ```

- You can select the column and the **toString( )** method, which lists in one string all of the field values in the column:

  ```
  select home_addr>>toString() from emps
  ```

Assignment Properties of Java-SQL Data Items

The values assigned to Java-SQL data items are derived ultimately from values constructed by Java-SQL methods in the Java VM. However, the logical representation of Java-SQL variables, parameters, and results is different from the logical representation of Java-SQL columns.

- Java-SQL *columns*, which are persistent, are Java serialized streams stored in the containing row of the table. They are stored values containing representations of Java instances.
- Java-SQL *variables*, *parameters*, and *function results* are transient. They do not actually contain Java-SQL instances, but instead contain references to Java instances contained in the Java VM.

These differences in representation give rise to differences in assignment properties as these examples illustrate.

- The Address constructor method with the **new** operator is evaluated in the Java VM. It constructs an **Address** instance and returns a reference to it. That reference is assigned as the value of Java-SQL variable *@A*:

```
declare @A Address, @AA Address, @A2 Address2Line,
   @AA2 Address2Line
select @A = new Address('432 Post Lane', '99444')
```

- Variable *@A* contains a reference to a Java instance in the Java VM. That reference is copied into variable *@AA*. Variables *@A* and *@AA* now reference the same instance.

```
select @AA=@A
```

- This assignment modifies the *zip* field of the **Address** referenced by *@A*. This is the same **Address** instance that is referenced by *@AA*. Therefore, the values of *@A.zip* and *@AA.zip* are now both '99222'.

```
select @A>>zip='99222'
```

- The **Address** constructor method with the **new** operator constructs an **Address** instance and returns a reference to it. However, since the target is a Java-SQL column, the SQL system serializes the **Address** instance denoted by that reference, and copies the serialized value into the new row of the *emps* table.

```
insert into emps
   values ('Don Green', new Address('234 Stone
   Road', '99777'), new Address2Line( ) )
```

The **Address2Line** constructor method operates the same way as the **Address** method, except that it returns a default instance rather than an instance with specified parameter values. The action taken is, however, the same as for the **Address** instance. The SQL system serializes the default **Address2Line** instance, and stores the serialized value into the new row of the *emps* table.

- The **insert** statement specifies no value for the mailing_addr column, so that column will be set to **null**, in the same manner as any other column whose value is not specified in an **insert**. This null value is generated entirely in SQL, and initialization of the *mailing_addr* column does not involve the Java VM at all.

```
insert into emps (name, home_addr)  values ('Frank Lee', @A)
```

The **insert** statement specifies that the value of the *home_addr* column is to be taken from the Java-SQL variable *@A*. That variable contains a reference to an **Address** instance in the Java VM. Since the target is a Java-SQL column, the SQL system serializes the **Address** instance denoted by *@A*, and copies the serialized value into the new row of the *emps* table.

- This statement inserts a new *emps* row for 'Bob Brown.' The value of the *home_addr* column is taken from the SQL variable *@A*. It is also a serialization of the Java instance referenced by *@A*.

```
insert into emps (name, home_addr)  values ('Bob Brown', @A)
```

- This update statement sets the zip field of the *home_addr* column of the 'Frank Lee' row to '99777.' This has no effect on the *zip* field in the 'Bob Brown' row, which is still '99444.'

```
update emps
   set home_addr>>zip = '99777'
   where name = 'Frank Lee'
```

- The Java-SQL column *home_addr* contains a serialized representation of the value of an **Address** instance. The SQL system invokes the Java VM to de-serialize that representation as a Java instance in the Java VM, and return a reference to the new deserialized copy. That reference is assigned to *@AA*. The deserialized **Address** instance that is referenced by *@AA* is entirely independent of both the column value and the instance referenced by *@A*.

```
select @AA = home_addr from emps where name = 'Frank Lee'
```

- This assignment modifies the zip field of the **Address** instance referenced by *@A*. This instance is a copy of the *home_addr* column of the 'Frank Lee' row, but is independent of that column value. The assignment therefore does not modify the zip field of the *home_addr* column of the 'Frank Lee' row.

```
select @A>>zip = '95678'
```

Datatype Mapping Between Java and SQL Fields

When you transfer data in either direction between the Java VM and Adaptive Server, you must take into account that the datatypes of the data items are different in each system. Adaptive Server automatically maps SQL items to Java items and vice versa according to the correspondence tables in “Datatype Mapping Between Java and SQL” on page 166.

Thus, SQL type char translates to Java type String, the SQL type binary translates to the Java type byte[], and so on.

- For the datatype correspondences from SQL to Java, char, varchar, and varbinary types of any length correspond to Java String or byte[ ] datatypes, as appropriate.
- For the datatype correspondences from Java to SQL:
  - The Java String and byte[ ] datatypes correspond to SQL varchar(255) and varbinary(255), where the maximum length value of 255 bytes is defined by Adaptive Server.
  - The Java BigDecimal datatype corresponds to SQL numeric(precision,scale), where precision and scale are defined by the user.

Since the maximum length values for varchar and varbinary are 255 bytes, the **Address** and **Address2Line** classes, *street*, *zip*, and *line2* fields, whose Java datatypes are all String, are treated in SQL as datatype varchar(255).

An expression whose datatype is a Java object type is converted to the corresponding SQL datatype only when the expression is used in a SQL context. For example, if the field *home_addr>>street* for employee ‘Smith’ is 260 characters, and begins ‘6789 Main Street ...:

```
select Misc.getStreet(home_addr>>street) from emps where name='Smith'
```

The expression in the **select** list passes the 260-character value of *home_addr>>street* to the **getStreet( )** method (without truncating it to 255 characters). The **getStreet( )** method then returns the 255-character string beginning ‘Main Street….’. That 255-character string is now an element of the SQL **select** list, and is, therefore, converted to the SQL datatype and (if need be) truncated to 255 characters.

Character Sets for Data and Identifiers

The character set for both Java program text and for Java String data is Unicode.

In Java program text installed in SQL, the Java identifiers used in the fully qualified names of visible classes or in the names of visible members can use only Latin characters and Arabic numerals.

Fields of Java-SQL classes can also contain Unicode data.

Subtypes in Java-SQL Data

Class subtypes allow you to use the substitution and method overloading characteristics of Java. A conversion from a class to one of its superclasses is a widening conversion; a conversion from a class to one of its subclasses is a narrowing conversion.

- Widening conversions are performed implicitly with normal assignments and comparisons. They are always successful, since every subclass instance is also an instance of the superclass.
- Narrowing conversions must be specified with explicit **convert** expressions. A narrowing conversion is successful only if the superclass instance is an instance of the subclass, or a subclass of the subclass. Otherwise, an exception occurs.

Widening Conversions

You do not need to use the **convert** function to specify a widening conversion. For example, since the **Address2Line** class is a subclass of the **Address** class, you can assign **Address2Line** values to **Address** data items. In the *emps* table, the *home_addr* column is an **Address** datatype and the *mailing_addr* column is an **Address2Line** datatype:

```
update emps
set home_addr = mailing_addr
where home_addr is null
```

For the rows fulfilling the **where** clause, the *home_addr* column contains an **Address2Line**, even though the declared type of *home_addr* is **Address**.

Such an assignment implicitly treats an instance of a class as an instance of a superclass of that class. The runtime instances of the subclass retain their subclass datatypes and associated data.

Narrowing Conversions

You must use the **convert** function to convert an instance of a class to an instance of a subclass of the class. For example:

```
update emps
   set mailing_addr = convert(Address2Line, home_addr)
   where mailing_addr is null
```

The narrowing conversions in the **update** statement cause an exception if they are applied to any *home_addr* column that contains an **Address** instance that is not an **Address2Line**. You can avoid such exceptions by including a condition in the **where** clause:

```
update emps
   set mailing_addr = convert(Address2Line, home_addr)
   where mailing_addr is null
   and home_addr>>getClass( )>>toString( ) = 'Address2Line'
```

The expression "home_addr>>getClass( )>>toString( )" invokes **getClass( )** and **toString( )** methods of the Java **Object** class. The **Object** class is implicitly a superclass of all classes, so the methods defined for it are available for all classes.

You can also use a **case** expression:

```
update emps
   set mailing_addr =
      case
         when home_addr>>getClass( )>>toString( )
           ='Address2Line'
         then convert(Address2Line, home_addr)
         else null
      end
   where mailing_addr is null
```

Runtime vs. Compile-Time Datatypes

Neither widening nor narrowing conversions modify the actual instance value or its runtime datatype; they simply specify the class to be used for the compile-time type. Thus, when you store **Address2Line** values from the *mailing_addr* column into the *home_address* column, those values still have the runtime type of **Address2Line**.

For example, the **Address** class and the **Address2Line** subclass both have the method **toString( )**, which returns a String form of the complete address data.

```
select name, home_addr>>toString( )  from emps
   where home_addr>>toString( ) not like '%Line2=[ ]'
```

For each row of *emps*, the declared type of the *home_addr* column is **Address**, but the runtime type of the *home_addr* value is either **Address** or **Address2Line**, depending on the effect of the previous **update** statement. For rows in which the runtime value of the *home_addr* column is an **Address**, the **toString( )** method of the **Address** class is invoked, and for rows in which the runtime value of the *home_addr* column is **Address2Line**, the **toString( )** method of the **Address2Line** subclass is invoked.

See "Null Values When Using the SQL convert Function" on page 47 for a description of null values for widening and narrowing conversions.

The Treatment of Nulls in Java-SQL Data

This section discusses the use of nulls in Java-SQL data items.

References to Fields and Methods of Null Instances

If the value of the instance specified in a field reference is null, then the field reference is null. Similarly, if the value of the instance specified in an instance method invocation is null, then the result of the invocation is null.

In Java, if you attempt to reference a field of a null instance, an exception is raised. Java in Adaptive Server does not follow this convention, allowing you to write **select** and other statements, even if some rows in *emps* contain null values for *home_addr*. For example:

```
select name, home_addr>>zip  from emps
   where home_addr>>zip in ('95123', '95125', '95128')
```

For rows whose *home_addr* column is null, the field reference *home_addr>>zip* is also null. The **where** clause is evaluated for each row of *emps*, including those rows in which the *home_addr* column is null.

Note, however, that this rule for field references with null instances only applies to field references in source (right-side) contexts, not to field references that are targets (left-side) of assignments or **set** clauses. For example:

```
update emps
   set home_addr>>zip = '99123'
   where name = 'Charles Green'
```

This **where** clause is obviously true for the 'Charles Green' row, so the **update** statement tries to perform the **set** clause. This raises an exception, since you cannot assign a value to a field of a null instance as the null instance has no field to which a value can be assigned. Thus, field references to fields of null instances are valid and return the null value in right-side contexts, and cause exceptions in left-side contexts.

The same considerations apply to invocations of methods of null instances, and the same rule is applied. For example, if we modify the previous example and invoke the **toString( )** method of the *home_addr* column:

```
select name, home_addr>>toString( )  from emps
   where home_addr>>toString( ) = 'Street=234 Stone
     Road ZIP= 99777'
```

If the value of the instance specified in an instance method invocation is null, then the result of the invocation is null. Hence, the **select** statement is valid here, whereas it raises an exception in Java.

Null Values as Arguments to Java-SQL Methods

Null parameter values are independent of the actions of the method for which they are an argument, but instead depend on the ability of the return datatype to deliver a null value.

You cannot pass the null value as a parameter to a Java scalar type method; Java scalar types are always non-nullable. However, Java object types can accept null values.

For the following Java-SQL class:

```
public class General implements java.io.Serializable {
   public static int identity1(int I) {return I;}
   public static java.lang.Integer identity2
     (java.lang.Integer I) {return I;}
   public static Address identity3 (Address A) {return A;}
   }
```

Consider these calls:

```
declare @I int
declare @A Address;

select @I = General.identity1(@I)
select @I = General.identity2(new java.lang.Integer(@I))
select @A = General.identity3(@A)
```

The values of both variable *@I* and variable *@A* are null, since values have not been assigned to them.

- The call of the **identity1( )** method raises an exception. The datatype of the parameter *@I* of **identity1( )** is the Java int type, which is scalar and has no null state. An attempt to pass a null valued argument to **identity1( )** raises an exception.
- The call of the **identity2( )** method succeeds. The datatype of the parameter of **identity2( )** is the Java class **java.lang.Integer**, and the **new** expression creates an instance of **java.lang.Integer** that is set to the value of variable *@I*.
- The call of the **identity3( )** method succeeds.

A successful call of **identity1( )** never returns a null result, since the return type has no null state. Successful calls of **identity2( )** and **identity3( )** can return null results.

Null Values When Using the SQL *convert* Function

You use the **convert** function to convert a Java object of one class to a Java object of a superclass or subclass of that class.

As shown in "Subtypes in Java-SQL Data" on page 42, the *home_addr* column of the *emps* table can contain values of both the **Address** class and the **Address2Line** class. In this example:

```
select name, home_addr>>street, convert(Address2Line, home_addr)>>line2,
   home_addr>>zip from emps
```

the expression "convert(Address2Line, home_addr)" specifies a datatype (**Address2Line**) and an expression (*home_addr*). At compile-time, the expression (*home_addr*) must be a subtype or supertype of the class (**Address2Line**). At runtime, the action of this **convert** invocation depends on whether the value of the expression is a class, subclass, or superclass:

- If the runtime value of the expression (*home_addr*) is the specified class (**Address2Line**) or its subclass, the value of the expression is returned, with the specified datatype (**Address2Line**).
- If the runtime value of the expression (*home_addr*) is a superclass of the specified class (**Address**), then a null is returned.

Adaptive Server evaluates the **select** statement for each row of the result. For each row:

- If the value of the *home_addr* column is an **Address2Line**, then **convert** returns that value, and the field reference extracts the *line2* field.

Hence, the results of the **select** shows the *line2* value for those rows whose *home_addr* column is an **Address2Line** and a null for those rows whose *home_addr* column is an **Address**. As described in "The Treatment of Nulls in Java-SQL Data" on page 45, the **select** also shows a null *line2* value for those rows in which the *home_addr* column is null.

Java-SQL String Data

In Java-SQL columns, fields of type String are stored as Unicode.

When a Java-SQL String field is assigned to a SQL data item whose type is char, varchar, nchar, nvarchar, or text, the Unicode data is converted to the character set of the SQL system. Conversion errors are specified by the **set char_convert** options.

When a SQL data item whose type is char, varchar, nchar, or text is assigned to a Java-SQL String field that is stored as Unicode, the character data is converted to Unicode. Undefined codepoints in such data cause conversion errors.

Zero-Length Strings

In Transact-SQL, a zero-length character string is treated as a null value, and the empty string ( ) is treated as a single space.

To be consistent with Transact-SQL, when a Java-SQL String value whose length is zero is assigned to an SQL data item whose type is char, varchar, nchar, nvarchar, or text, the Java-SQL String value is replaced with a single space.

For example:

```
1> declare @s varchar(20)
2> select @s = new  java.lang.String()
3> select @s, char_length(@s)
4> go
```

(1 row affected)

```
----------------  ----------------
```

1

If the zero-length Java-SQL String value was assigned to the SQL data item as a zero-length string, that zero-length value would be treated in SQL as a SQL null, and when assigned to a Java-SQL String, the Java-SQL String would be a Java null.

Type and Void Methods

Java methods (both instance and static) are either type methods or void methods. In general, type methods return a value of the result type, and void methods perform some action(s) and return nothing.

For example, in the **Address** class:

- The **toString( )** method is a *type method* whose type is String.
- The **removeLeadingBlanks( )** method is a *void method*.
- The **Address** constructor method is a *type method* whose type is the **Address** class.

As in Java, you invoke type methods as functions and use the **new** keyword when invoking a constructor method:

```
insert into emps
   values ('Don Green', new Address('234 Stone Road', '99777'),
      new Address2Line( ) )

select name, home_addr>>toString( ) from emps
   where home_addr>>toString( )  like '%Baker%'
```

The **removeLeadingBlanks( )** method of the **Address** class is a void instance method that modifies the *street* and *zip* fields of a given instance. You can invoke **removeLeadingBlanks( )** for the *home_addr* column of each row of the *emps* table. For example:

```
update emps
 set home_addr =
   home_addr>>removeLeadingBlanks( )
```

removeLeadingBlanks( ) removes the leading blanks from the *street* and *zip* fields of the *home_addr* column. The Transact-SQL **update** statement does not provide a framework or syntax for such an action. It simply replaces column values.

Java Void Instance Methods

To use the "update-in-place" actions of Java void instance methods in the SQL system, Java in Adaptive Server treats a call of a Java void instance method as follows:

For a void instance method **M( )** of an instance *CI* of a class **C**, written "**CI.M(...)**":

- In SQL, the call is treated as a type method call. The result type is implicitly class **C**, and the result value is a reference to *CI*. That reference identifies the instance *CI* after the actions of the void instance method call.
- In Java, this call is a void method call, which performs its actions and returns no value.

For example, you can invoke the **removeLeadingBlanks( )** method for the *home_addr* column of selected rows of the *emps* table as follows:

```
update emps
   set home_addr = home_addr>>removeLeadingBlanks( )
   where home_addr>>removeLeadingBlanks( )>>street like “123%”
```

1. In the **where** clause, “home_addr>>removeLeadingBlanks( )” calls the **removeLeadingBlanks( )** method for the *home_addr* column of a row of the *emps* table. **removeLeadingBlanks( )** strips the leading blanks from the *street* and *zip* fields of a copy of the column. The SQL system then returns a reference to the modified copy of the *home_addr* column. The subsequent field reference:

   ```
   home_addr>>removeLeadingBlanks( )>>street
   ```

   returns the *street* field that has the leading blanks removed. The references to *home_addr* in the **where** clause are operating on a copy of the column. This evaluation of the **where** clause does *not* modify the *home_addr* column.
2. The **update** statement performs the **set** clause for each row of *emps* in which the **where** clause is true.
3. On the right-side of the **set** clause, the invocation of “home_addr>>removeLeadingBlanks( )” is performed as it was for the **where** clause: **removeLeadingBlank( )** strips the leading blanks from *street* and *zip* fields of that copy. The SQL system then returns a reference to the modified copy of the *home_addr* column.
4. The **Address** instance denoted by the result of the right-side of the **set** clause is serialized and copied into the column specified on the left-side of the **set** clause: the result of the expression on the right-side of the **set** clause is a copy of the *home_addr* column in which the leading blanks have been removed from the *street* and *zip* fields. The modified copy is then assigned back to the *home_addr* column as the new value of that column.

The expressions of the right and left side of the right-side clause are independent, as is normal for the **update** statement.

The following **update** statement shows an invocation of a void instance method of the *mailing_addr* column on the on the right side of the **set** clause being assigned to the *home_address* column on the left side.

```
update emps
   set home_addr = mailing_addr>>removeLeadingBlanks( )
   where ...
```

In this **set** clause, the void method **removeLeadingBlanks( )** of the *mailing_addr* column yields a reference to a modified copy of the **Address2Line** instance in the *mailing_addr* column. The instance denoted by that reference is then serialized and assigned to the *home_addr* column. This action updates the *home_addr* column; it has no effect on the *mailing_addr* column.

Java Void Static Methods

With Adaptive Server Version 12, you cannot invoke a void static method using a simple SQL **execute** command. Rather, you must place the invocation of the void static method in a **select** statement.

For example, suppose that a Java class **C** has a void static method **M(...)**, and assume that **M( )** performs an action you want to invoke in SQL. For example, **M( )** can use JDBC calls to perform a series of SQL statements that have no return values, such as **create** or **drop**, that would be appropriate for a void method.

You must invoke the void static method in a **select** command, such as:

```
select C.M(...)
```

To allow void static methods to be invoked using a **select**, void static methods are treated in SQL as returning a value of datatype int with a value of null.

Equality and Ordering Operations

You can use equality and ordering operators when you use Java in the database; however, you cannot:

- Reference Java-SQL data items in ordering operations.
- Reference Java-SQL data items in equality operations if, at runtime, their representation is greater than 255 bytes.
- Use the **order by** clause, which requires that you determine the sort order.
- Make direct comparisons using the “>”, “<”, “<=”, or “>=” operator.

These equality operations are allowed in JCS:

- Use of the **distinct** keyword, which is defined in terms of equality of rows, including Java-SQL columns.
- Direct comparisons using the “=” and “!=” operators.
- Use of the **union** operator (not **union all**), which eliminates duplicates, and requires the same kind of comparisons as the **distinct** clause.
- Use of the **group by** clause, which partitions the rows into sets with equal values of the grouping column.

Call-by-Reference for Java Methods

Adaptive Server does not have a defined order for evaluating operands of comparisons and other operations. Instead, Adaptive Server evaluates each query and chooses an evaluation order based on the most rapid rate of execution.

This section describes how different evaluation orders affect the outcome when you pass columns or variables and parameters as arguments. The examples in this section use the following Java-SQL class:

```
public class Utility implements java.io.Serializable {
       public static int F (Address A) {
          if (A.zip.length( ) > 5) return 0;
          else {A.zip = A.zip + "-1234"; return 1;}
       }
       public static int G (Address A) {
          if (A.zip.length( ) > 5) return 0;
          else {A.zip = A.zip + "-1234"; return 1;}
       }
}
```

Columns

In general, avoid using methods or value-returning contexts that modify their arguments. Where there are multiple invocations of the same or different methods, the order of evaluation can affect the outcome.

For example, in this example:

```
select *  from emp E
   where Utility.F(E.home_addr) > Utility.F(E.home_addr)
```

the **where** clause passes the same *home_addr* column in two different method invocations. Consider the evaluation of the **where** clause for a row whose *home_addr* column has a 5-character zip, such as “95123.”

Adaptive Server can initially evaluate either the left or right side of the comparison. After the first evaluation completes, the second is processed. Because it executes faster this way, Adaptive Server may let the second invocation see the modifications of the argument made by the first invocation.

In the example, the first invocation chosen by Adaptive Server returns 1, and the second returns 0. If the left operand is evaluated first, the comparison is 1>0, and the **where** clause is true; if the right operand is evaluated first, the comparison is 0>1, and the **where** clause is false.

Variables and Parameters

Similarly, the order of evaluation can affect the outcome when passing variables and parameters as arguments.

Consider the following statements:

```
declare @A Address
declare @Order  varchar(20)

select @A = new Address('95444', '123 Port Avenue')
select case when Utility.F(@A)>Utility.G(@A)
       then 'Left' else 'Right' end
select @Order = case when utility.F(@A) > utility.G(@A)
       then 'Left'  else 'Right' end
```

The new **Address** has a five-character zip code field. When the **case** expression is evaluated, depending on whether the left or right operand of the comparison is evaluated first, the comparison is either 1>0 or 0>1, and the *@Order* variable is set to 'Left' or 'Right' accordingly.

As for column arguments, the expression value depends on the evaluation order. Depending on whether the left or right operand of the comparison is evaluated first, the resulting value of the *zip* field of the **Address** instance referenced by *@A* is either "95444-4321" or "95444-1234."

Static Variables in Java-SQL Classes

A Java variable that is declared static is associated with the Java class, rather than with each instance of the class. The variable is allocated once for the entire class.

For example, you might include a static variable in the **Address** class that specifies the recommended limit on the length of the *Street* field:

```
public class  Address   implements java.io.Serializable {
   public static int recommendedLimit;
   public String street;
   public String zip;
   // ...
}
```

You can specify that a static variable is **final**, which indicates that it is not updatable:

```
public static final int recommendedLimit;
```

Otherwise, you can update the variable.

You reference a static variable the same way as a dynamic variable—by qualifying the variable name with an instance of the class:

```
select Address.recommendedLimit = 20
if Address.recommendedLimit < 50
   select Address.recommendedLimit = Address.recommended_limit + 5
```

Values assigned to non-final static variables are accessible only within the current session.

Java Classes in Multiple Databases

You can store Java classes of the same name in different databases in the same Adaptive Server system. This section describes how you can use these classes.

Scope

When you install a Java class or set of classes, they are installed in the current database. When you dump or load a database, the Java-SQL classes that are currently installed in that database are always included—even if classes of the same name exist in other databases in the Adaptive Server system.

You can install Java classes with the same name in different databases. These synonymous classes can be:

- Identical classes that have been installed in different databases.
- Different classes that are intended to be mutually compatible. Thus, a serialized value generated by either class is acceptable to the other.
- Different classes that are intended to be "upward" compatible. That is, a serialized value generated by one of the classes should be acceptable to the other, but not vice versa.
- Different classes that are intended to be mutually incompatible; for example, a class named **Sheet** designed for supplies of paper, and other classes named **Sheet** designed for supplies of linen.

Cross-Database References

You can reference classes in one database from another database.

For example, assume the following configuration:

- The **Address** class is installed in *db1* and *db2*.
- The *emps* table has been created in both *db1* with owner Smith, and in *db2*, with owner Jones.

In these examples, the current database is *db1*. You can invoke a join or a method across databases. For example:

- A **join** across databases might look like this:

```
declare @count  int
select @count(*)
```

```
from db2.Jones.emps, db1.Smith.emps
where db2.Jones.emps.home_addr>>zip =
   db1.Smith.emps.home_addr>>zip
```

- A method invocation across databases might look like this:

```
select db2.Jones.emps.home_addr>>toString( )
   from db2.Jones.emps
   where db2.Jones.emps.name = 'John Stone'
```

In these examples, instance values are not transferred. Fields and methods of an instance contained in *db2* are merely referenced by a routine in *db1*. Thus, for across-database joins and method invocations:

- *db1* need not contain an **Address** class.
- If *db1* does contain an **Address** class, it can have completely different properties than the **Address** class in *db2*.

Inter-Class Transfers

You can assign an instance of a class in one database to an instance of a class of the same name in another database. Instances created by the class in the source database are transferred into columns or variables whose declared type is the class in the current (target) database.

You can insert or update from a table in one database to a table in another database. For example:

```
insert into db1.Smith.emps select * from
   db2.Jones.emps

update db1.Smith.emps
   set home_addr = (select db2.Jones.emps.home_addr
      from db2.Jones.emps
   where db2.Jones.emps.name =
      db1.Smith.emps.name)
```

You can insert or update from a variable in one database to a another database. (The following fragment is in a stored procedure on *db2*.) For example:

```
declare @home_addr Address
select @home_addr = new Address('94608', '222 Baker
   Street')
insert into db1.Janes.emps(name, home_addr)
   values ('Jone Stone', @home_addr)
```

In these examples, instance values are transferred between databases. You can:

- Transfer instances between two local databases.
- Transfer instances between a local database and a remote database.
- Transfer instances between a SQL client and an Adaptive Server.
- Replace classes using **install** and **update** statements or **remove** and **update** statements.

In an inter-class transfer, the Java serialization is transferred from the source to the target.

Passing Inter-Class Arguments

You can pass arguments between classes of the same name in different databases.When passing inter-class arguments:

- A Java-SQL column is associated with the version of the specified Java class in the database that contains the column.
- A Java-SQL variable (in Transact-SQL) is associated with the version of the specified Java class in the current database.
- A Java-SQL intermediate result of class **C** is associated with the version of class **C** in the same database as the Java method that returned the result.
- When a Java instance value *JI* is assigned to a target variable or column, or passed to a Java method, *JI* is converted from its associated class to the class associated with the receiving target or method.

Temporary and Work Databases

All rules for Java classes and databases also apply to temporary databases and the model database:

- Java-SQL columns of temporary tables contain byte string serializations of the Java instances.
- A Java-SQL column is associated with the version of the specified class in the temporary database.

You can install Java classes in a temporary database, but they will persist only as long as the temporary database persists.

The simplest way to provide Java classes for reference in temporary databases is to install Java classes in the model database. They are then present in any temporary database derived from the model.

Sample Java Classes

This section shows the simple Java classes that this chapter uses to illustrate Java in Adaptive Server. You can also find these classes and their Java source code in *$SYBASE/$SYBASE_ASE/sample/JavaSql*. (UNIX) or *%SYBASE%\Ase-12_0\sample\JavaSql* (Windows NT).

This is the **Address** class:

```
//
// Copyright (c) 1999
// Sybase, Inc
// Emeryville, CA 94608
// All Rights Reserved
//
/**
* A simple class for address data, to illustrate using a Java class
* as a SQL datatype.
*/

public class  Address implements java.io.Serializable  {

/**
* The street data for the address.
* @serial A simple String value.
*/
   public String street;

/**
* The zipcode data for the address.
* @serial A simple String value.
*/
   String zip;

/** A default constructor.
*/
   public Address ( ) {
      street = "Unknown";
      zip = "None";
   }
/**
* A constructor with parameters
* @param S   a string with the street information
* @param Z   a string with the zipcode information
*/
   public Address (String S, String Z) {
      street = S;
```

```
      zip = Z;
   }
/**
* A method to return a display of the address data.
* @returns a string with a display version of the address data.
*/
   public String toString( ) {
      return  "Street= " + street +  "  ZIP= " + zip;
   }
/**
* A void method to remove leading blanks.
* This method uses the static method
* <code>Misc.stripLeadingBlanks</code>.
*/
   public void removeLeadingBlanks( ) {
   street = Misc.stripLeadingBlanks(street);
   zip = Misc.stripLeadingBlanks(street);
   }
}
```

This is the **Address2Line** class, which is a subclass of the **Address** class:

```
//
// Copyright (c) 1999
// Sybase, Inc
// Emeryville, CA 94608
// All Rights Reserved
//
/**
* A subclass of the Address class that adds a seond line of address data,
* <p>This is a simple subclass to illustrate using a Java subclass
* as a SQL datatype.
*/
public class Address2Line extends Address implements java.io.Serializable  {

/**
* The second line of street data for the address.
* @serial a simple String value
*/
   String line2;
/**
* A default constructor
*/
   public Address2Line ( ) {
      street = "Unknown";
      line2 = " ";
      zip = "None";
```

```
   }
/**
* A constructor with parameters.
* @param S a string with the street information
* @param L2 a string with the second line of address data
* @param Z a string with the zipcode information
*/
public Address2Line (String S, String L2, String Z) {
   street = S;
   line2 = L2;
   zip = Z;
}

/**
* A method to return a display of the address data
* @returns a string with a display version of the address data
*/

public String toString( ) {
   return "Street= " + street + " Line2= " + line2 +  " ZIP= " + zip;
}

/**
* A void method to remove leading blanks.
* This method uses the static method
* <code>Misc.stripLeadingBlanks</code>.
*/

   public void removeLeadingBlanks( ) {
      line2  = Misc.stripLeadingBlanks(line2);
      super.removeLeadingBlanks( );
   }
}
```

The **Misc** class contains sets of miscellaneous routines:

```
//
// Copyright (c) 1999
// Sybase, Inc
// Emeryville, CA 94608
// All Rights Reserved
//
/**
* A non-instantiable class with miscellaneous static methods
* that illustrate the use of Java methods in SQL.
*/
```

```
public class Misc{

/**
* The Misc class contains only static methods and cannot be instantiated.
*/

private Misc( ) { }

/**
* Removes leading blanks from a String
*/
   public static String stripLeadingBlanks(String s) {
      if (s == null) return null;
      for (int scan=0; scan<s.length( ); scan++)
         if (!java.lang.Character.isWhitespace(s.charAt(scan) ))
            break;
         } else if (scan == s.length( )){
            return  "";
         } else return s.substring(scan);
         }
      }
   }
   return  "";
}
/**
* Extracts the street number from an address line.
* e.g., Misc.getNumber("   123 Main Street") == 123
*    Misc.getNumber("   Main Street")  == 0
*    Misc.getNumber("") == 0
*    Misc.getNumber("   123   ") ==  123
*    Misc.getNumber("   Main 123 ")  == 0
* @param s a string assumed to have address data
* @return a string with the extracted street number
*/

   public static int getNumber (String s) {
      String stripped = stripLeadingBlanks(s);
      if (s==null) return -1;
      for(int right=0; right < stripped.length( ); right++){
         if (!java.lang.Character.isDigit(stripped.charAt(right))) {
            break;
         } else if (right==0){
            return 0;
         } else {
            return java.lang.Integer.parseInt
               (stripped.substring(0, right), 10);
```

```
         }
         }
         return -1;
      }

/**
* Extract the "street" from an address line.
* e.g.,  Misc.getStreet("   123 Main Street") ==  "Main Street"
*  Misc.getStreet("   Main Street")  ==   "Main Street"
*  Misc.getStreet("") ==  ""
*  Misc.getStreet("   123   ") ==  ""
*  Misc.getStreet("   Main 123 ")  == "Main 123"
* @param s a string assumed to have address data
* @return a string with the extracted street name
*/
   public static String getStreet(String s) {
      int left;
      if (s==null) return null;
      for (left=0; left<s.length( ); left++){
         if(java.lang.Character.isLetter(s.charAt(left))) {
            break;
         } else if (left == s.length( )) {
            return "";
         } else {
            return s.substring(left);
         }
      }
      return "";
   }
}
```

CHAPTER 4

Data Access Using JDBC

This chapter describes how to use Java Database Connectivity (JDBC) to access data.

These topics are discussed:

Overview

JDBC provides a SQL interface for Java applications. If you want to access relational data from Java, you must use JDBC calls.

You can use JDBC with the Adaptive Server SQL interface in either of two ways:

- *JDBC on the client* – Java client applications can make JDBC calls to Adaptive Server using the Sybase jConnect JDBC driver.
- *JDBC on the server* – Java classes installed in the database can make JDBC calls to the database using the JDBC driver internal to Adaptive Server.

The use of JDBC calls to perform SQL operations is essentially the same in both contexts.

This chapter provides sample classes and methods that describe how you might perform SQL operations using JDBC. These classes and methods are not intended to serve as templates, but as general guidelines.

JDBC Concepts and Terminology

JDBC is a Java API and a standard part of the Java class libraries that control basic functions for Java application development. The SQL capabilities that JDBC provides are similar to those of ODBC and dynamic SQL.

The following sequence of events is typical of a JDBC application:

1 Create a *Connection* object – Call the **getConnection( )** class method of the **DriverManager** class to create a *Connection* object. This establishes a database connection.

2 Generate a *Statement* object – Use the *Connection* object to generate a *Statement* object.

3 Pass a SQL statement to the *Statement* object – If the statement is a query, this action returns a *ResultSet* object.

The *ResultSet* object contains the data returned from the SQL statement, but provides it one row at a time (similar to the way a cursor works).

4 Loop over the rows of the results set – Call the **next( )** method of the *ResultSet* object to:

- Advance the current row (the row in the result set that is being exposed through the *ResultSet* object) by one row.
- Return a Boolean value (true/false) to indicate whether there is a row to advance to.

5 For each row, retrieve the values for columns in the *ResultSet* object – use the **getInt( )**, **getString( )**, or similar method to identify either the name or position of the column.

Differences Between Client- and Server-Side JDBC

The difference between JDBC on the client and in the database server is in how a connection is established with the database environment.

- *Client-side JDBC* – Requires the Sybase jConnect JDBC driver to establish a connection. The connection is established by passing arguments to the **DriverManager.getConnection( )** method. The database environment is an external application from the perspective of the Java client application.
- *Server-side JDBC* – When JDBC is used within the database server, a connection already exists. A value of "jdbc:default:connection" is passed to **DriverManager.getConnection( )**, which provides the JDBC application the ability to work within the current user connection. This is a safe and efficient operation because the client application has already passed the database security to establish the connection.

  You can write JDBC classes to run both at the client and at the server by employing a single conditional statement for constructing the URL.

  An external connection requires the machine name and port number, while the internal connection requires one of these values:

  - **jdbc:default:connection**
  - **jdbc:sybase:ase**
  - **jdbc:default**

Connections and Permissions

- *Connection defaults* – From server-side JDBC, only the first call to **getConnection("jdbc:default:connection")** creates a new connection with the default values.

  Subsequent calls return a wrapper of the current connection with all connection properties unchanged.

- *Access permissions* – Like all Java classes in the database, classes containing JDBC statements can be accessed by any user. There is no equivalent of the **grant execute** statement that grants permission to execute procedures, and there is no need to qualify the name of a class with the name of its owner.

- *Execution permissions* – Java classes are executed with the permissions of the connection executing them. This behavior is different from that of stored procedures, which execute with the permissions of the owner.

Using JDBC to Access Data

Java applications that hold some or all classes in the database have significant advantages over traditional SQL stored procedures.

This section describes how you can use JDBC to perform the typical operations of a SQL application. The examples are extracted from the class **JDBCExamples**, which is described in "The JDBCExamples Class" on page 77 and in *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE%\Ase-12_0\sample\JavaSql* (Windows NT).

JDBCExamples illustrates the basics of a user interface and shows the internal coding techniques for SQL operations.

Overview of the *JDBCExamples* Class

The **JDBCExamples** class uses the **Address** class described in "Sample Java Classes" on page 11. To execute these examples on your machine, install the **Address** class on the server and include it in the Java CLASSPATH of the jConnect client.

You can call the methods of **JDBCExamples** from either a jConnect client or Adaptive Server.

Note You must create or drop stored procedures from the jConnect client. The Adaptive Server internal driver does not support **create procedure** and **drop procedure** statements.

JDBCExamples class methods perform the following SQL operations:

- Create and drop an example table, *xmp*:

```
create table xmp (id int, name varchar(50), home Address)
```

- Create and drop a sample stored procedure, *inout*:

```
create procedure inout @id int, @newname varchar(50),
   @newhome Address, @oldname varchar(50) output, @oldhome
   Address output as

select @oldname = name, @oldhome = home from xmp
   where id=@id
update xmp set name=@newname, home = @newhome
   where id=@id
```

- Insert a row into the *xmp* table.
- Select a row from the *xmp* table.
- Update a row of the *xmp* table.
- Call the stored procedure *inout*, which has both input parameters and output parameters of datatypes **java.lang.String** and **Address**.

JDBCExamples operates only on the *xmp* table and *inout* procedure.

The *main( )* and *serverMain( )* Methods

JDBCExamples has two primary methods:

- **main( )** – is invoked from the command line of the jConnect client.
- **serverMain( )** – performs the same actions as **main( )**, but is invoked within Adaptive Server.

All actions of the **JDBCExamples** class are invoked by calling one of these methods, using a parameter to indicate the action to be performed.

Using *main( )*

You can invoke the **main( )** method from a jConnect command line as follows:

```
java JDBCExamples  "server-name:
   port-number?user=user-name&password=password" action
```

You can determine *server-name* and *port-number* from your interfaces file. *user-name* and *password* are your user name and password. If you omit **&password=***password*, the default is the empty password. Here are two examples:

```
"antibes:4000?user=smith&password=1x2x3"
"antibes:4000?user=sa"
```

Make sure that you enclose the parameter in quotation marks.

The *action* parameter can be **create table**, **create procedure**, **insert**, **select**, **update**, or **call**. It is case insensitive.

You can invoke **JDBCExamples** from a jConnect command line to create the table *xmp* and the stored procedure *inout* as follows:

```
java JDBCExamples "antibes:4000?user=sa" CreateTable
java JDBCExamples "antibes:4000?user=sa" CreateProc
```

You can invoke **JDBCExamples** for **insert**, **select**, **update**, and **call** actions as follows:

```
java JDBCExamples “antibes:4000?user=sa” insert
java JDBCExamples “antibes:4000?user=sa” update
java JDBCExamples “antibes:4000?user=sa” call
java JDBCExamples “antibes:4000?user=sa” select
```

These invocations display the message “Action performed.”

To drop the table *xmp* and the stored procedure *inout*, enter:

```
java JDBCExamples “antibes:4000?user=sa” droptable
java JDBCExamples “antibes:4000?user=sa” dropproc
```

Using *serverMain( )*

Note Because the server-side JDBC driver does not support **create procedure** or **drop procedure**, create the table *xmp* and the example stored procedure *inout* with client-side calls of the **main( )** method before executing these examples. Refer to “Overview of the JDBCExamples Class” on page 70.

After creating *xmp* and *inout*, you can invoke the **serverMain( )** method as follows:

```
select JDBCExamples.serverMain('insert')
go
select JDBCExamples.serverMain('select')
go
select JDBCExamples.serverMain('update')
go
select JDBCExamples.serverMain('call')
go
```

Note Server-side calls of **serverMain( )** do not require a *server-name:port-number* parameter; Adaptive Server simply connects to itself.

Obtaining a JDBC Connection: the *Connecter( )* Method

Both **main( )** and **serverMain( )** call the **connecter( )** method, which returns a JDBC *Connection* object. The *Connection* object is the basis for all subsequent SQL operations.

Both **main( )** and **serverMain( )** call **connecter( )** with a parameter that specifies the JDBC driver for the server- or client-side environment. The returned *Connection* object is then passed as an argument to the other methods of the **JDBCExamples** class. By isolating the connection actions in the **connecter( )** method, **JDBCExamples**' other methods are independent of their server- or client-side environment.

Routing the Action to Other Methods: the *doAction( )* Method

The **doAction( )** method routes the call to one of the other methods, based on the *action* parameter.

doAction( ) has the *Connection* parameter, which it simply relays to the target method. It also has a parameter *locale*, which indicates whether the call is server- or client-side. *Connection* raises an exception if either **create procedure** or **drop procedure** is invoked in a server-side environment.

Executing Imperative SQL Operations: the *doSQL( )* Method

The **doSQL( )** method performs SQL actions that require no input or output parameters such as **create table**, **create procedure**, **drop table**, and **drop procedure**.

doSQL( ) has two parameters: the *Connection* object and the SQL statement it is to perform. **doSQL( )** creates a JDBC *Statement* object and uses it to execute the specified SQL statement.

Executing an *update* Statement: the *UpdateAction( )* Method

The **updateAction( )** method performs a Transact-SQL **update** statement. The **update** action is:

```
String sql = "update xmp set name = ?, home = ? where id = ?";
```

It updates the *name* and *home* columns for all rows with a given *id* value.

The **update** values for the *name* and *home* column, and the *id* value, are specified by parameter markers (?). **updateAction( )** supplies values for these parameter markers after preparing the statement, but before executing it. The values are specified by the JDBC **setString( )**, **setObject( )**, and **setInt( )** methods with these parameters:

- The ordinal parameter marker to be substituted
- The value to be substituted

For example:

```
pstmt.setString(1, name);
pstmt.setObject(2, home);
pstmt.setInt(3, id);
```

After making these substitutions, **updateAction( )** executes the **update** statement.

To simplify **updateAction( )**, the substituted values in the example are fixed. Normally, applications would compute the substituted values or obtain them as parameters.

Executing a *select* Statement: the *selectAction( )* Method

The **selectAction( )** method executes a Transact-SQL **select** statement:

```
String sql = "select name, home from xmp where id=?";
```

The **where** clause has a parameter marker (?) for the row to be selected. Using the JDBC **setInt( )** method, **selectAction( )** supplies a value for the parameter marker after preparing the SQL statement:

```
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setInt(1, id);
```

selectAction( ) then executes the **select** statement:

```
ResultSet rs = pstmt.executeQuery();
```

Note For SQL statements that return no results, use **doSQL( )** and **updateAction( )**. They execute SQL statements with the **executeUpdate( )** method.

For SQL statements that do return results, use the **executeQuery( )** method, which returns a JDBC *ResultSet* object.

The *ResultSet* object is similar to a SQL cursor. Initially, it is positioned before the first row of results. Each call of the **next( )** method advances the *ResultSet* object to the next row, until there are no more rows.

selectAction( ) requires that the *ResultSet* object have exactly one row. The **selecter( )** method invokes the next method, and checks for the case where *ResultSet* has no rows or more than one row.

```
if (rs.next()) {
   name = rs.getString(1);
   home = (Address)rs.getObject(2);
   if (rs.next()) {
      throw new Exception("Error:  Select returned multiple rows");
   } else { // No action
   }
} else { throw new Exception("Error:  Select returned no rows");
}
```

In the above code, the call of methods **getString( )** and **getObject( )** retrieve the two columns of the first row of the result set. The expression "(Address)rs.getObject(2)" retrieves the second column as a Java object, and then coerces that object to the **Address** class. If the returned object is not an **Address**, then an exception is raised.

selectAction( ) retrieves a single row and checks for the cases of no rows or more than one row. An application that processes a multiple row *ResultSet* would simply loop on the calls of the **next( )** method, and process each row as for a single row.

Calling a SQL Stored Procedure: the *callAction( )* Method

The **callAction( )** method calls the stored procedure *inout*:

```
create proc inout @id int, @newname varchar(50), @newhome Address,
       @oldname varchar(50) output,  @oldhome Address output as

   select @oldname = name, @oldhome = home from xmp where id=@id
   update xmp set name=@newname, home = @newhome where id=@id
```

This procedure has three input parameters (*@id*, *@newname*, and *@newhome*) and two output parameters (*@oldname* and *@oldhome*). **callAction( )** sets the name and home columns of the row of table *xmp* with the ID value of *@id* to the values *@newname* and *@newhome*, and returns the former values of those columns in the output parameters *@oldname* and *@oldhome*.

The *inout* procedure illustrates how to supply input and output parameters in a JDBC call.

callAction( ) executes the following call statement, which prepares the call statement:

```
CallableStatement cs = con.prepareCall("{call inout (?, ?, ?, ?, ?)}");
```

All of the parameters of the call are specified as parameter markers (?).

callAction( ) supplies values for the input parameters using JDBC **setInt( )**, **setString( )**, and **setObject( )** methods that were used in the **doSQL( )**, **updatAction( )**, and **selectAction( )** methods:

```
cs.setInt(1, id);
cs.setString(2, newName);
cs.setObject(3, newHome);
```

These **set** methods are not suitable for the output parameters. Before executing the call statement, **callAction( )** specifies the datatypes expected of the output parameters using the JDBC **registerOutParameter( )** method:

```
cs.registerOutParameter(4, java.sql.Types.VARCHAR);
cs.registerOutParameter(5,com.sybase.jdbc.Param.JAVA_OBJECT);
```

callAction( ) then executes the call statement and obtains the output values using the same **getString( )** and **getObject( )** methods that the **selectAction( )** method used:

```
int res = cs.executeUpdate();
String oldName = cs.getString(4);
Address oldHome = (Address)cs.getObject(5);
```

The *JDBCExamples* Class

```
// An example class illustrating the use of JDBC facilities
// with the Java in Adaptive Server feature.
//
// The methods of this class perform a range of SQL operations.
// These methods can be invoked either from a Java client,
// using the main method, or from the SQL server, using
// the internalMain method.
//
import java.sql.*;             // JDBC
public class JDBCExamples {
{
```

The *main( )* Method

```
// The main method, to be called from a client-side command line
//
    public static void main(String args[]) {
        if (args.length!=2) {
            System.out.println("\n Usage:    "
                 + "java ExternalConnect server-name:port-number
              action ");
            System.out.println("  The action is connect, createtable,
          " + "createproc, drop, "
          + "insert, select, update, or call \n" );
            return;
        }
        try{
           String server = args[0];
           String action = args[1].toLowerCase();
           Connection con = connecter(server);
           String workString = doAction( action, con, client);
           System.out.println("\n" + workString + "\n");
        }  catch (Exception e) {
           System.out.println("\n Exception: ");
           e.printStackTrace();
        }
    }
```

The *internalMain( )* Method

```
// A JDBCExamples method equivalent to 'main',
```

```
// to be called from SQL or Java in the server

    public static String internalMain(String action) {
        try {
           Connection con = connecter("default");
           String workString = doAction(action, con, server);
           return workString;
        }  catch ( Exception e ) {
             if (e.getMessage().equals(null)) {
               return "Exc: " + e.toString();
             } else {
               return "Exc - " + e.getMessage();
             }
        }
    }
```

The *connecter( )* Method

```
// A JDBCExamples method to get a connection.
// It can be called from the server with argument 'default',
// or from a client, with an argument that is the server name.

public static  Connection connecter(String server)
       throws Exception, SQLException, ClassNotFoundException  {

       String forName="";
       String url="";

       if (server=="default") {  // server connection to current server
       forName = "sybase.asejdbc.ASEDriver";
       url = "jdbc:default:connection";
       } else if (server!="default") {  //client connection to server
       forName= "com.sybase.jdbc.SybDriver";
       url = "jdbc:sybase:Tds:"+ server;
       }

       String user = "sa";
       String password = "";

       // Load the driver
       Class.forName(forName);
       // Get a connection
       Connection con = DriverManager.getConnection(url,
       user, password);
       return con;
```

```
    }
```

The *doAction( )* Method

```
// A JDBCExamples method to route to the 'action' to be performed

    public static String doAction(String action, Connection con,
          String locale)
       throws Exception {

        String createProcScript =
             " create proc inout @id int, @newname varchar(50),
          @newhome Address, "
           + "   @oldname varchar(50) output,  @oldhome Address
          output as "
           + " select @oldname = name, @oldhome = home from xmp
          where id=@id "
           + " update xmp set name=@newname, home = @newhome
          where id=@id ";
        String createTableScript =
       " create  table xmp (id int, name varchar(50),
          home Address)" ;

        String dropTableScript = "drop   table xmp  ";
        String dropProcScript = "drop  proc inout ";
      String insertScript = "insert  into xmp "
      + "values (1, 'Joe Smith', new Address('987 Shore',
      '12345'))";

        String workString = "Action (" + action + ) ;
        if (action.equals("connect")) {
             workString += "performed";
        } else if (action.equals("createtable")) {
             workString += doSQL(con, createTableScript );
        } else if (action.equals("createproc")) {
             if (locale.equals(server))  {
                 throw new exception (CreateProc cannot be performed
              in the server);
             }  else {
                 workString += doSQL(con, createProcScript );
             }
        } else if (action.equals("droptable")) {
             workString += doSQL(con, dropTableScript );
        } else if (action.equals("dropproc")) {
             if (locale.equals(server))  {
```

```
                throw new exception (CreateProc cannot be performed
             in the server);
             }  else {
                 workString += doSQL(con, dropProcScript );
             }
        } else if (action.equals("insert")) {
             workString += doSQL(con, insertScript );
        } else if (action.equals("update")) {
             workString += updateAction(con);
        } else if (action.equals("select")) {
             workString += selectAction(con);
        } else if (action.equals("call")) {
             workString += callAction(con);
        } else { return "Invalid action:  " + action ;
        }
        return workString;
    }
```

The *doSQL( )* Method

```
// A JDBCExamples method to execute an SQL statement.

    public static String doSQL (Connection con, String action)
           throws Exception {

        Statement stmt = con.createStatement();
        int res = stmt.executeUpdate(action);
        return "performed";
 }
```

The *updateAction( )* Method

```
// A method that updates a certain row of the 'xmp' table.
// This method illustrates prepared statements and parameter markers.

    public static String updateAction(Connection con)
           throws Exception {

        String sql = "update xmp set name = ?, home = ? where id = ?";
        int id=1;
        Address home = new Address("123 Main", "98765");
        String name = "Sam Brown";
        PreparedStatement pstmt = con.prepareStatement(sql);
```

```
        pstmt.setString(1, name);
        pstmt.setObject(2, home);
        pstmt.setInt(3, id);
        int res = pstmt.executeUpdate();
        return "performed";
    }
```

The *selectAction( )* Method

```
//  A JDBCExamples method to retrieve a certain row
//  of the 'xmp' table.
//  This method illustrates prepared statements, parameter markers,
//  and result sets.

    public static String selectAction(Connection con)
           throws Exception {

        String sql = "select name, home from xmp where id=?";
        int id=1;
        Address home = null;
        String name = "";
        String street = "";
        String zip = "";
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
           name = rs.getString(1);
           home = (Address)rs.getObject(2);
           if (rs.next()) {
              throw new Exception("Error:  Select returned
              multiple rows");
           } else { // No action
           }
        } else { throw new Exception("Error:  Select returned no rows");
        }
        return "- Row with id=1:  name("+ name + )
                + " street(" + home.street + ) zip("+ home.zip + );
```

The *callAction( )* Method

```
// A JDBCExamples method to call a stored procedure,
// passing input and output parameters of datatype String
```

```
 // and Address.
 // This method illustrates callable statements, parameter markers,
 // and result sets.

    public static String callAction(Connection con)
           throws Exception {
       CallableStatement cs  = con.prepareCall("{call inout
      (?, ?, ?, ?, ?)}");
       int id = 1;
       String newName = "Frank Farr";
       Address newHome = new Address("123 Farr Lane", "87654");
       cs.setInt(1, id);
       cs.setString(2, newName);
       cs.setObject(3, newHome);
       cs.registerOutParameter(4, java.sql.Types.VARCHAR);
       cs.registerOutParameter(5, com.sybase.jdbc.Param.JAVA_OBJECT);
       int res = cs.executeUpdate();
       String oldName = cs.getString(4);
       Address oldHome = (Address)cs.getObject(5);
       return "- Old values of row with id=1: name("+oldName+ )
      street(" + oldHome.street + ")  zip("+ oldHome.zip + );
    }
}
```

CHAPTER 5

XML in the Database

This chapter uses examples to describe how you can use Java tools to access Extensible Markup Language (XML) documents in Adaptive Server.

These topics are discussed:

Introduction

Like Hypertext Markup Language (HTML), XML is a markup language and a subset of Standardized General Markup Language (SGML). XML, however, is more complete and disciplined, and it allows you to define your own application-oriented markup tags. These properties make XML particularly suitable for data interchange.

You can generate XML-formatted documents from data stored in Adaptive Server and, conversely, store XML documents, and data extracted from them, in Adaptive Server. Many of the XML tools needed to generate and process XML documents are written in Java. Java in Adaptive Server provides a good base for XML-SQL applications using both universal and application-specific tools.

This chapter first provides a general discussion of XML and how you can use XML in the Adaptive Server database. It then presents a series of examples that you can use as guidelines for using XML in your Adaptive Server database.

Source Code and Javadoc

The source code for the Java classes described in this chapter is available in *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE%\Ase-12_0\sample\JavaSql* (Windows NT), which also contains Javadoc-generated HTML pages with the specifications of the referenced packages, classes, and methods.

References

This chapter presents a overview of XML. For detailed information, refer to these Web documents.

- World Wide Web Consortium (W3C), at http://www.w3.org
- W3C, Document Object Model (DOM), at http://www.w3.org/DOM/
- W3C, Extensible Markup Language (XML™), at http://www.w3.org/XML/
- W3C, Extensible Stylesheet Language (XSL), at http://www.w3.org/TR/WD-xsl/

- Sun Microsystems, Inc, Java™ Project X Technology Release 1, at http://developer.java.sun.com/developer/earlyAccess/xml/index.html
- Megginson Technologies, SAX 1.0: The Simple API for XML, at http://www.megginson.com/SAX/

An Overview of XML

XML is a markup language and subset of SGML. It was created to provide functionality that goes beyond that of HTML for Web publishing and distributed document processing.

XML is less complex than SGML, but more complex and flexible than HTML. Although XML and HTML can usually be read by the same browsers and processors, XML has characteristics that make it better able to share documents:

- XML documents possess a strict phrase structure that makes it easy to find and access data. For example, opening tags of all elements must have a corresponding closing tag, for example, <p>A paragraph.<\p>.
- XML lets you develop and use tags that distinguish different types of data, for example, customer numbers or item numbers.
- XML lets you create an application-specific document type, which makes it possible to distinguish one kind of document from another.
- XML documents allow different views of the XML data. XML documents contain only markup and content; they do not contain formatting instructions. Formatting instructions are normally provided on the client using Extensible Style Language (XSL) specifications.

A Sample XML Document

The sample Order document is designed for a purchase order application. Customers submit orders, which are identified by a date and a customer ID. Each order item has an item ID, an item name, a quantity, and a unit designation.

It might display on screen like this:

ORDER

Date: July 4, 1999

Customer ID: 123

Customer Name: Acme Alpha

Items:

Item ID	Item Name	Quantity
987	Coupler	5
654	Connector	3 dozen
579	Clasp	1

A possible XML representation of the data for the order is:

```
<?xml version="1.0"?>
<Order>
 <Date>1999/07/04</Date>
 <CustomerId>123</CustomerId>
 <CustomerName>Acme Alpha</CustomerName>
   <Item>
 <ItemId> 987</ItemId>
 <ItemName>Coupler</ItemName>
 <Quantity>5</Quantity>
 </Item>
<Item>
 <ItemId>654</ItemId>
 <ItemName>Connector</ItemName>
 <Quantity unit="12">3</Quantity>
 </Item>
<Item>
 <ItemId>579</ItemId>
 <ItemName>Clasp</ItemName>
 <Quantity>1</Quantity>
 </Item>
</Order>
```

The XML document for the order data consists of these parts:

- The XML declaration, <?xml version=“1.0”?>, which identifies Order as an XML document.

  XML documents are represented as character data. In each document, the character encoding (character set) is specified, either explicitly or implicitly. To explicitly specify the character set, include it in the XML declaration. For example:

```
<?xml version=”1.0” encoding=”ISO-8859-1”>
```

If you do not include the character set in the XML declaration, the default, which is UTF8, is used. For example:

```
<?xml version=”1.0”?>
```

Note When the default character sets of the client and server differ, Adaptive Server bypasses normal character set translations so that the declared character set continues to match the actual character set. See “Character Sets and XML Data” on page 91.

- User-created element tags such as <Order>…</Order>, <CustomerId>…</CustomerId>, <Item>….</Item>. In XML documents, all opening tags must have a corresponding closing tag.
- Text data such as “Acme Alpha,” “Coupler,” and “579.”
- Attributes embedded in element tags such as <Quantity unit = “12”>. This kind of coding allows you the flexibility to customize elements.

A document with these parts, and with the element tags strictly nested, is called a **well-formed XML document**. Note that in the example above element tags describe the data they contain, and the document contains no formatting instructions.

XML Document Types

A **Document Type Definition** (DTD) defines the structure of a class of XML documents, making it possible to distinguish between classes. A DTD is a list of element and attribute definitions unique to a class. Once you have set up a DTD, you can reference a DTD in another document or embed the DTD in the XML document.

Here is another example of an XML document:

```
<?xml version="1.0"?>
 <Info>
 <OneTag>1999/07/04</OneTag>
 <AnotherTag>123</AnotherTag>
 <LastTag>Acme Alpha</LastTag>
   <Thing>
   <ThingId> 987</ThingId>
   <ThingName>Coupler</ThingName>
   <Amount>5</Amount>
```

```
   <Thing/>
   <Thing>
 <ThingId>654</ThingI
 <ThingName>Connecter</ThingNam
 </Thing>
<Thing>
   <ThingId>579</ThingId>
   <ThingName>Clasp</ThingName>
   <Amount>1</Renew>
 </Thing>
 </Info>
```

This example, called Info, is a well-formed document and has the same structure and data as the XML Order document. Nonetheless, it would not be recognized by a processor designed for Order documents because each have different DTDs.

The DTD for XML Order documents is:

```
<!ELEMENT Order (Date, CustomerId, CustomerName,
    Item+)>
 <!ELEMENT Date (#PCDATA)>
 <!ELEMENT CustomerId (#PCDATA)>
 <!ELEMENT CustomerName (#PCDATA)>
 <!ELEMENT Item (ItemId, ItemName, Quantity)>
 <!ELEMENT ItemId (#PCDATA)>
 <!ELEMENT ItemName (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 <!ATTLIST Quantity units CDATA #IMPLIED>
```

This DTD specifies that:

- An order consists of these required elements: a date, a customer ID, a customer name, and one or more items. “+” indicates one or more items. These items are required. A question mark indicates an optional element (for example, “CustomerName?”). An asterisk indicates that an element can occur zero or more times (for example, “Item*”).
- Elements defined by “(#PCDATA)” are character text.
- The “<ATTLIST…>” definition specifies that quantity elements have a “units” attribute; the “#IMPLIED” specification indicates that the “units” attribute is optional.

The character text of XML documents is not constrained. For example, there is no way to specify that the text of a quantity element should be numeric, and thus the following would be valid:

```
<Quantity unit=”Baker’s dozen”>three</Quantity>
<Quantity unit=”six packs”>plenty</Quantity>
```

Restrictions on the text of elements are handled by applications that process XML data.

An XML's DTD must follow the <?xml version="1.0"?> instruction. You can either include the DTD within your XML document, or you can reference an external DTD.

- To reference a DTD externally, use something like this:

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM  "Order.dtd”>
 <Order>
…
 </Order>
```

- Here's how an embedded DTD might look:

```
<?xml version="1.0"?>
 <!DOCTYPE Order [
 <!ELEMENT Order (Date, CustomerId, CustomerName,
 Item+)>
 <!ELEMENT Date (#PCDATA)
 <!ELEMENT CustomerId (#PCDATA)>
 <!ELEMENT CustomerName (#PCDATA)>
 <!ELEMENT Item (ItemId, ItemName, Quantity)>
 <!ELEMENT ItemId (#PCDATA)>
 <!ELEMENT ItemName (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 <!ATTLIST Quantity units CDATA #IMPLIED>

]>
 <Order>
      <Date>1999/07/04</Date>
      <CustomerId>123</CustomerId>
      <CustomerName>Acme Alpha</CustomerName>

   <Item>
      …
 </Item>
 </Order>
```

DTDs are not required for XML documents. However, a **valid XML document** has a DTD and conforms to that DTD.

XSL: Formatting XML Information

You can use XSL to format XML documents. XSL specifications (stylesheets) consist of a set of rules that define the transformation of an XML document into either an HTML document or a different XML document:

- XSL specifications that transform an XML document into HTML can specify normal HTML formatting details in the output HTML.
- XSL specifications that transform an XML document into another XML document can map the input XML document to an output XML document with different element names and phrase structure.

You can create your own stylesheets for the display of particular classes for particular applications. XSL is normally used with presentation applications rather than with applications for data interchange or storage.

Character Sets and XML Data

If the declared character sets of your client and server differ, you must take care when declaring the character set of your XML documents.

Every XML document has a character set value. If that encoding is not declared in the XML declaration, the default value of UTF8 is assumed. The XML processor, when parsing the XML data, reads this value and handles the data accordingly. When the default character set of the client and server differ, Adaptive Server bypasses normal character set conversions to ensure that the declared character set and the actual character set remain the same.

- If you introduce an XML document into the database by providing the complete text in the values clause of an **insert** statement, Adaptive Server translates the entire SQL statement into the server's character set before processing the insertion. This is the way Adaptive Server normally translates character text, and you must make sure that the declared character set of the XML document matches that of the server.
- If you introduce an XML document into the database using **writetext** or Open Client CT-Library or Open Client DB-Library programs, Adaptive Server recognizes the XML document from the XML declaration and does *not* translate the character set to that of the server.
- If you read an XML document from the database, Adaptive Server does *not* translate the character set of the data to that of the client, thus preserving the integrity of the XML document.

Using XML in the Adaptive Server Database

To use XML documents for data interchange in Adaptive Server, you must be able to store XML documents or the data that they contain in the database. To determine how best to accomplish this, consider the following:

- *Mapping and storage*: What sort of correspondence between XML documents and SQL data is most suitable for your system?
- *Client or Server Considerations*: Should the mapping take place on the client or the server?
- *Accessing XML in SQL*: How do you want to access the elements of an XML document in SQL?

The rest of this section discusses each of these considerations; the remainder of the chapter provides these classes and methods you can use with XML:

- A simple example to illustrate the basics of data storage and exchange of XML documents
- A generalized example that you can customize for your own XML documents

Mapping and Storage

There are three basic ways to store XML data in Adaptive Server: **element storage**, **document storage**, or **hybrid storage**, which is a mixture of both.

- *Element storage* – In this method, you extract data elements from an XML document and store them as data rows and columns in Adaptive Server.

  For example, using the XML Order document, you can create SQL tables with columns for the individual elements of an order: *Date*, *CustomerId*, *CustomerName*, *ItemId*, *ItemName*, *Quantity*, and *Units*. You can then manage that data in SQL with normal SQL operations:

  - To produce an XML document for Order data contained in SQL, retrieve the data, and assemble an XML document with it.
  - To store an XML document with new Order data, extract the elements of that document, and update the SQL tables with that data.
- *Document storage* – In this method, you store an entire XML document in a single SQL column.

For example, using the Order document, you can create one or more SQL tables having a column for Order documents. The datatype of that column could be:

- SQL text, or
- A generic Java class designed for XML documents, or
- A Java class designed specifically for XML Order documents

- *Hybrid storage* – In this method, you store an XML document in a SQL column, and also extract some of its data elements into separate columns for faster and more convenient access.

  Again, using the Order example, you can create SQL tables as you would for document storage, and then include (or later add) one or more columns to store elements extracted from the Order documents.

Advantages and Disadvantages of Storage Options

Each storage option has advantages and disadvantages. You must choose the option or options best for your operation.

- If you use element storage, all of the data from the XML document is available as normal SQL data that you can query and update using SQL operations. However, element storage has the overhead of assembling and disassembling the XML documents for interchange.
- Document storage eliminates the need for assembling and disassembling the data for interchange. However, you need to use Java methods to reference or update the elements of the XML documents while they are in SQL, which is slower and less convenient than the direct SQL access of element storage.
- Hybrid storage balances the advantages of element storage and document storage, but has the cost and complexity of redundant storage of the extracted data.

Client or Server Considerations

This chapter describes Java methods for assembling and disassembling an XML document and referencing or updating its elements. You can execute Java methods either on the client or on the server, which is a consideration for element storage and hybrid storage. Document storage involves little or no processing of the document.

- Element storage – If you map individual elements of an XML document to SQL data, in most cases, the XML document is larger than the SQL data. It is generally more efficient to assemble and disassemble the XML document on the client and transfer only the SQL data between the client and the server.
- Hybrid storage – If you store both the complete XML document and extracted elements, then it is generally more efficient to extract the data from the server, rather than transfer it from the client.

Accessing XML in SQL

This chapter discusses three applications of XML in SQL. These applications are organized in three layers:

- *Transact-SQL statements* such as **insert**, **select**, and **update** for referencing SQL columns and variables that contain XML documents. These SQL operations use Java classes and methods to manipulate the XML documents.
- *Java classes* to contain XML documents and to access and update the elements of those documents. There is an application-specific class for the Order document type and a general class for arbitrary SQL result sets.
- An *XML parser*, which is used by the Java classes to analyze and manipulate XML documents.

The Java classes that are used in this chapter to demonstrate XML applications are **JXml**, **OrderXml**, and **ResultSetXml**.

- **JXml** stores and parses XML. It does not validate XML documents. It is designed as a base class for subclasses that:
  - Validate specific XML document types
  - Provide application-oriented methods

  OrderXml and **ResultSetXml** are two such subclasses.
- The **OrderXml** classes used to illustrate support for an application-specific XML document type. **OrderXml** validates Order documents for the Order DTD. You can use **OrderXml** methods to reference and update elements of the Order document.

- **ResultSetXml** represents SQL result sets. The **ResultSetXml** constructor validates the ResultSet document for the ResultSet DTD. **ResultSetXml** methods are used to reference and update elements of the ResultSet document.

  The **ResultSetXml** class illustrates support for a general XML document type capable of representing arbitrary SQL data.

"The OrderXml Class for Order Documents" on page 97 and "The ResultSetXml Class for Result Set Documents" on page 116 describe these classes and their methods and parameters. For Javadoc HTML pages with detailed specifications for the classes and for source code, refer to *$SYBASE/$SYBASE_ASE/sample/JavaSql* (UNIX) or *%SYBASE%\Ase-12_0\sample\JavaSql* (Windows NT).

XML Parsers

You can analyze XML documents and extract their data using SQL character-string operations such as **substring**, **charindex**, and **patindex**. However, it is more efficient to use Java in SQL and tools written in Java such as XML parsers.

XML parsers can:

- Check that a document is well-formed and valid.
- Handle character-set issues.
- Generate a Java representation of a document's parse tree.
- Build or modify a document's parse tree.
- Generate a document's text from its parse tree.

Many XML parsers are available with a free license or are in the public domain. They normally implement two standard interfaces: the Simple API for XML (SAX) and the Document Object Model (DOM).

- *SAX* is an interface for parsing. It specifies input sources, character sets, and routines to handle external references. While parsing, it generates events so that user routines can process the document incrementally, and it returns a DOM object that is the parse tree of the document.
- *DOM* is an interface for the parse tree of an XML document. It provides facilities for stepping through and assembling a parse tree.

Applications that use the SAX and DOM interfaces are portable across XML parsers.

A Simple Example for a Specific Result Set

This section provides a simple example that demonstrates how you can store XML documents or the data that they contain in an Adaptive Server database.

The example in this section, the XML Order document type, is designed for a specific purchase-order application, and the Java methods created for it assume a specific set of SQL tables for storing purchase order data.

For a more generalized example, applicable to a range of SQL result sets, see "A Customizable Example for Different Result Sets" on page 112:

The *OrderXml* Class for Order Documents

The example in this section uses the **OrderXml** class and its methods for basic operations on XML Order documents. The source code and Javadoc specifications for **OrderXml** are in *$SYBASE/$SYBASE_ASE/sample/JavaSql*.

OrderXml is a subclass of the **JXml** class, which is specialized for XML Order documents. The **OrderXml** constructor validates the document for the Order DTD. Methods of the **OrderXml** class support referencing and updating the elements of the Order document.

- Constructor: **OrderXml(String)**

  Validates that the *String* argument contains a valid XML Order document, and then constructs an **OrderXml** object containing that document. For example, "doc" is a Java string variable containing an XML Order document, perhaps one read from a file:

```
jcs.xml.order.OrderXml  ox = new jcs.xml.order.OrderXml(doc);
```

- Constructor: **OrderXml(date, customerId, dtdOption, server)**

  The parameters are all String.

  This method assumes a set of SQL tables containing Order data. The method uses JDBC to execute a SQL query that retrieves Order data for the given *date* and *customerId*. The method then assembles an XML Order document with the data.

  The *server* parameter identifies the Adaptive Server on which to execute the query.

  - If you invoke the method in a client environment, specify the server name.

- If you invoke the method in Adaptive Server (in a SQL statement or in **isql**), specify either an empty string or the string "jdbc:default:connection," which indicates that the query should be executed on the current Adaptive Server

The *dtdOption* parameter indicates whether you want the generated Order to contain the DTD or to reference it externally.

For example:

```
jcs.xml.order.OrderXml ox = new OrderXml("990704", "123",
   "external", "antibes:4000?user=sa");
```

- **void order2Sql(String ordersTableName, String server)**

  Extracts the elements of the Order document and stores them in a SQL table created by the **createOrdertable( )** method. *ordersTableName* is the name of the target table. The *server* parameter is as described for the **OrderXml** constructor. For example, if *ox* is a Java variable of type OrderXml:

```
ox.order2Sql("current_orders", "antibes:4000?user=sa");
```

  This call extracts the elements of the Order document contained in *ox*, and uses JDBC to insert the extracted elements into rows and columns of the table named *current_orders*.

- **static void createOrderTable(String ordersTableName, String server)**

  Creates a SQL table with columns suitable for storing Order data: *customer_id*, *order_date*, *item_id*, *quantity*, and *unit*. *ordersTableName* is the name of the new table. The *server* parameter is as described for the **OrderXml** constructor. For example:

```
jcs.xml.order.OrderXml.createOrderTable
   ("current_orders", "antibes:4000?user=sa");
```

- **String getOrderElement(String elementName)**

  elementName is "Date," "CustomerId," or "CustomerName." The method returns the text of the element. For example, if *ox* is a Java variable of type **OrderXml**:

```
String customerId = ox.getOrderElement("CustomerId");
String customerName = ox.getOrderElement("CustomerName");
String date = ox.getOrderElement("Date");
```

- **void setOrderElement(String elementName, String newValue)**

elementName is as described for **getOrderElement( )**.The method sets that element to **newValue**. For example, if *ox* is a Java variable of type **OrderXml**:

```
ox.setOrderElement("CustomerName", "Acme Alpha Consolidated");
ox.setOrderElement("CustomerId", "987a");
ox.setOrderElement("Date", "1999/07/05");
```

- **String getItemElement(int itemNumber, String elementName)**

  itemNumber is the index of an item in the order. *elementName* is "ItemId," "ItemName," or "Quantity." The method returns the text of the item. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  String itemId = ox.getItemElement(2, "ItemId");
  String itemName = ox.getItemElement(2, "ItemName");
  String quantity = ox.getItemElement(2, "Quantity");
  ```

- **void setItemElement(int itemNumber, String elementName, String newValue)**

  itemNumber and *elementName* are as described for the **getItemElement** method. **setItemElement** sets the element to *newValue*. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.setItemElement(2, "ItemId", "44");
  ox.setItemElement(2, "ItemName", "cord");
  ox.setItemElement(2, "Quantity", "3");
  ```

- **String getItemAttribute(int itemNumber, elementName, attributeName)**

  itemNumber and *elementName* are described as for **getItemElement( )**. *elementName* and *attributeName* are both String. *attributeName* must be "unit." The method returns the text of the unit attribute of the item.

  Note Since the Order documents currently have only one attribute, the *attributeName* parameter is unnecessary. It is included to illustrate the general case, for example, if *ox* is a Java variable of type **OrderXml**:

  ```
  String itemId = ox.getItemAttribute(2, "unit");
  ```

- **void setItemAttribute (int itemNumber, elementName, attributeName, newValue)**

itemNumber, *elementName*, and *attributeName* are as described for **getItemAttribute( )**. *elementName*, *attributeName*, and *newValue* are String. The method sets the text of the unit attribute of the item to *newValue*. For example, if *ox* is a Java variable of type **OrderXml**:

```
ox.setItemAttribute(2, “unit”, “13”);
```

- **void appendItem(newItemId, newItemName, newQuantity, newUnit)**

  The parameters are all String. The method appends a new item to the document, with the given element values. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.appendItem(“77”, “spacer”, “5”, “12”);
  ```

- **void deleteItem(int itemNumber)**

  itemNumber is the index of an item in the order. The method deletes that item. For example, if *ox* is a Java variable of type **OrderXml**:

  ```
  ox.deleteItem(2);
  ```

Creating and Populating SQL Tables for Order Data

In this section we create several tables. These tables are designed to contain data from XML Order documents, so that we can demonstrate techniques for element, document, and hybrid data storage.

Tables for Element Storage

The following SQL statements create SQL tables *customers*, *orders*, and *items*, whose columns correspond with the elements of the XML Order documents.

```
create table customers
 (customer_id varchar(5) not null unique,
 customer_name varchar(50) not null)

create table orders
 (customer_id varchar(5) not null,
 order_date datetime not null,
 item_id varchar(5) not null,
 quantity int not null,
 unit smallint default 1)

create table items
 (item_id varchar(5) unique,
 item_name varchar(20))
```

These tables need not to have been specifically created to accommodate XML Order documents.

The following SQL statements populate the tables with the data in the example XML Order document (see “A Sample XML Document” on page 86):

```
insert into customers values("123", "Acme Alpha")

insert into orders values ("123", "1999/05/07",
 "987", 5, 1)

insert into orders values ("123", "1999/05/07",
 "654", 3, 12)

insert into orders values ("123", "1999/05/07",
 "579", 1, 1)

insert into items values ("987", "Widget")

insert into items values ("654",
 "Medium connecter")

insert into items values ("579",
 "Type 3 clasp")
```

Use **select** to retrieve the Order data from the tables:

```
select order_date as Date, c.customer_id as CustomerId,
 customer_name as CustomerName,
 o.item_id as ItemId, i.item_name as ItemName,
 quantity as Quantity, o.unit as unit
 from customers c, orders o, items i
 where c.customer_id=o.customer_id and
o.item_id=i.item_id
```

Date	CustomerId	CustomerName	ItemId	ItemName	Quantity	Unit
July 4 1999	123	Acme Alpha	987	Coupler	5	1
July 4 1999	123	Acme Alpha	654	Connector	3	12
July 4 1999	123	Acme Alpha	579	Clasp	1	1

Tables for Document and Hybrid Storage

The following SQL statement creates a SQL table for storing complete XML Order documents, either with or without extracted elements (for hybrid storage).

```
create table order_docs
 (id char(10) unique,
 customer_id varchar(5) null,  --  For an
   extracted “CustomerId” element
```

```
order_doc jcs.xml.order.OrderXml)
```

Using the Element Storage Technique

This section describes the element storage technique for bridging XML and SQL.

- “Composing Order Documents from SQL Data” on page 102 discusses the composition of an XML Order document from SQL data.
- “Decomposing Data from an XML Order into SQL” on page 103 discusses the decomposition of an XML Order document to SQL data.

Composing Order Documents from SQL Data

In this example, Java methods generate an XML Order document from the SQL data in the tables created in “Creating and Populating SQL Tables for Order Data” on page 100.

A constructor method of the **OrderXml** class maps the data. An call of that constructor might be:

```
new jcs.xml.order.OrderXml("990704", "123",
 “external”, "antibes:4000?user=sa");
```

This constructor method uses internal JDBC operations to:

- Execute a SQL query for the Order data
- Generate an XML Order document with the data
- Return the **OrderXml** object that contains the Order document

You can invoke the **OrderXml** constructor in the client or the Adaptive Server.

- If you invoke the **OrderXml** constructor in the client, the JDBC operations that it performs use jConnect to connect to the Adaptive Server and perform the SQL query. It then reads the result set of that query and generates the Order document on the client.
- If you invoke the **OrderXml** constructor in the Adaptive Server, the JDBC operations that it performs use the native JDBC driver to connect to the current Adaptive Server and perform the SQL query. It then reads the result set and generates the Order document in the Adaptive Server.

Generating an Order on the Client

Designed to be implemented on the client, **main( )** invokes the constructor of the **OrderXml** class to generate an XML Order from the SQL data. That constructor executes a **select** for the given date and customer id, and assembles an XML Order document from the result.

```
import java.io.*;
import jcs.util.*;
public class Sql2OrderClient {
   public static void main (String args[]) {
      try{
         jcs.xml.order.Order order =
            new jcs.xml.order.OrderXml("990704", "123",
            “external”, "antibes:4000?user=sa");
         FileUtil.string2File("Order-sql2Order.xml",
            order.getXmlText());
      } catch (Exception e) {
      System.out.println("Exception:");
      e.printStackTrace();
      }
   }
 }
```

Generating an Order on the Server

Designed for the server environment, the following SQL script invokes the constructor of the **OrderXml** class to generate an XML Order from the SQL data:

```
declare @order jcs.xml.order.OrderXml
select @order =
 new jcs.xml.order.OrderXml('990704', '123',
 'external', '')
insert into order_docs (id, order_doc) values(“3”,
 @order)
```

Decomposing Data from an XML Order into SQL

In this section, you extract elements from an XML Order document and store them in the rows and columns of the SQL Orders tables. The examples illustrate this procedure in both server and client environments.

You decompose the elements using the Java method **order2Sql( )** of the **OrderXml** class. Assume that *xmlOrder* is a Java variable of type OrderXml:

```
xmlOrder.order2Sql(“orders_received”, “antibes:4000?user=sa”);
```

The **order2Sql( )** call extracts the elements of the XML Order document contained in variable *xmlOrder*, and then uses JDBC operations to insert that data into the SQL table *orders_received*. You can call this method on the client or on Adaptive Server:

- Invoked from the client, **order2Sql( )** extracts the elements of the XML Order document in the client, uses jConnect to connect to the Adaptive Server, and then uses Transact-SQL **insert** to place the extracted data into the table.
- Invoked from the server, **order2Sql( )** extracts the elements of the XML Order document in the Adaptive Server, uses the native JDBC driver to connect to the current Adaptive Server, and then use Transact-SQL **insert** to place the extracted data into the table.

Decomposing the XML Document on the Client

Invoked from the client, the **main( )** method of the **Order2SqlClient** class creates a table named *orders_received* with columns suitable for Order data. It then extracts the elements of the XML Order contained in the file *Order.xml* into rows and columns of *orders_received*. It performs these actions with calls to the static method **OrderXml.createOrderTable( )** and the instance method **order2Sql( )**.

```
import jcs.util.*;
import jcs.xml.order.*;
import java.io.*;
import java.sql.*;
import java.util.*;
public class Order2SqlClient {
   public static void main (String args[]) {
      try{
         String xmlOrder =
            FileUtil.file2String("order.xml");
         OrderXml.createOrderTable("orders_received",
            "antibes:4000?user=sa");
         xmlOrder.order2Sql("orders_received",
            "antibes:4000?user=sa");
      } catch (Exception e) {
      System.out.println("Exception:");
      e.printStackTrace();
      }
   }
 }
```

Decomposing the XML Document on the Server

Invoked from the server, the following SQL script invokes the **OrderXml** constructor to generate an XML Order document from the SQL tables, and then invokes the method **OX.sql2Order( )**, which extracts the Order data from the generated XML and inserts it into the *orders_received* table.

```
declare @xmlorder OrderXml
select @xmlorder = new OrderXml('19990704', '123',
 'external', '')
select  @xmlorder>>order2Sql('orders_received', '')
```

Using the Document Storage Technique

When using the document storage technique, you store a complete XML document in a single SQL column.This approach avoids the cost of mapping the data between SQL and XML when documents are stored and retrieved, but access to the stored elements can be slow and inconvenient.

Storing XML Order Documents in SQL Columns

This section provides examples of document storage from the client and from the server.

Inserting an Order Document from a Client File

The following command-line call is representative of how you can insert XML data into Adaptive Server from a client file. It copies the contents of the *Order.xml* file (using the **–I** parameter) to the Adaptive Server and executes the SQL script (using the **–Q** parameter) using the contents of *Order.xml* as the value of the question-mark (*?*) parameter.

```
java jcs.util.FileUtil -A putstring -I "Order.xml" \
 -Q "insert into order_docs (id, order_doc)      \
   values ('1', new jcs.xml.order.OrderXml(?)) "   \
 –S "antibes:4000?user=sa"
```

Note The constructor invocation **new jcs.xml.order.OrderXml(?)** validates the XML Order document.

Inserting a Generated Order Document on the Server

Executed on the server, the following SQL command generates an XML Order document from SQL data, and immediately inserts the generated XML document into the column of the *order_docs* table.

```
insert into order_docs (ID, order_doc)
   select “2”,  new jcs.xml.order.OrderXml("990704", "123",
      "external", "")
```

Accessing the Elements of Stored XML Order Documents

We have created a table named *order_docs*, with a column named *order_doc*. The datatype of the *order_doc* column is **OrderXml**, which is a Java class that contains an XML Order document.

The **OrderXml** class contains several instance methods that let you reference and update elements of the XML Order document. They are described in “The OrderXml Class for Order Documents” on page 97. This section uses these methods to update the Order document.

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM  "Order.dtd">
 <Order>
 <Date>1999/07/04</Date>
 <CustomerId>123</CustomerId>
 <CustomerName>Acme Alpha</CustomerName>
<Item>
   <ItemId> 987</ItemId>
   <ItemName>Coupler</ItemName>
   <Quantity>5</Quantity>
 </Item>
<Item>
   <ItemId>654</ItemId>
   <ItemName>Connecter</ItemName>
   <Quantity unit="12">3</Quantity>
 </Item>
<Item>
   <ItemId>579</ItemId>
   <ItemName>Clasp</ItemName>
   <Quantity>1</Quantity>
 </Item>
 </Order>
```

Each XML Order document has exactly one *Date*, *CustomerId*, and *CustomerName*, and zero or more *Item*s, each of which has an *ItemId*, *ItemName*, and *Quantity*.

Client Access to Order Elements

The **main( )** method of the **OrderElements** class is executed on the client. It reads the *Order.xml* file into a local variable, and constructs an *OrderXml* document from it. The method then extracts the "header" elements (*Date*, *CustomerId*, and*CustomerName*) and the elements of the first Item of the Order, prints those elements, and finally updates those elements of the Order with new values.

```
import java.io.*;
import jcs.util.*;
public class OrderElements {
   public static void main ( String[] args) {
      try{

      String xml = FileUtil.file2String("Order.xml");
      jcs.xml.order.OrderXml ox =
         new jcs.xml.order.OrderXml(xml);

      // Get the header elements
      String cname = ox.getOrderElement("CustomerName");
      String cid   = ox.getOrderElement("CustomerId");
      String date = ox.getOrderElement("Date");

      // Get the elements for item 1 (numbering from 0)
      String iName1 = ox.getItemElement(1, "ItemName");
      String iId1 = ox.getItemElement(1, "ItemId");
      String iQ1 = ox.getItemElement(1, "Quantity");
      String iU = ox.getItemAttribute(1,  "Quantity", "unit");
      System.out.println("\nBEFORE UPDATE: ")
      System.out.println("\n   "+date+ "  "+ cname + " " +
         cid);
      System.out.println("\n   "+ iName1+" "+iId1+" "
         + iQ1 + " " + iU + "\n");

      //  Set the header elements
      ox.setOrderElement("CustomerName", "Best Bakery"
      ox.setOrderElement("CustomerId", "531");
      ox.setOrderElement("Date", "1999/07/31");

      //  Set the elements for item 1 (numbering from 0)
      ox.setItemElement(1, "ItemName", "Flange");
      ox.setItemElement(1, "ItemId", "777");
      ox.setItemElement(1, "Quantity","3");
      ox.setItemAttribute(1,  "Quantity", "unit", "13");

      //Get the updated header elements
      cname = ox.getOrderElement("CustomerName");
      cid   = ox.getOrderElement("CustomerId");
      date = ox.getOrderElement("Date");
```

```
      //  Get the updated elements for item 1
         (numbering from 0)
      iName1 = ox.getItemElement(1, "ItemName");
      iId1 = ox.getItemElement(1, "ItemId");
      iQ1 = ox.getItemElement(1, "Quantity");
      iU = ox.getItemAttribute(1,  "Quantity", "unit");

      System.out.println("\nAFTER UPDATE: ");
      System.out.println("\n   "+date+ "  "+ cname + " " +
         cid);
      System.out.println("\n   "+ iName1+" "+iId1+" "
         + iQ1 + " " + iU + "\n");

      //Copy the updated document to another file
      FileUtil.string2File("Order-updated.xml",
         ox.getXmlText())

      } catch (Exception e) {
      System.out.println("Exception:");
      e.printStackTrace();
      }
   }
}
```

After implementing the methods in **OrderElements**, the Order document stored in the file *Order-updated.xml* is:

```
<?xml version="1.0"?>
 <!DOCTYPE Order SYSTEM 'Order.dtd'>
 <Order>
 <Date>1999/07/31</Date>
 <CustomerId>531</CustomerId>
 <CustomerName>Best Bakery</CustomerName>
 <Item>
   <ItemId> 987</ItemId>
   <ItemName>Coupler</ItemName>
   <Quantity>5</Quantity>
 </Item>
 <Item>
   <ItemId>777</ItemId>
   <ItemName>Flange</ItemName>
   <Quantity unit="13">3</Quantity>
 </Item>
 <Item>
   <ItemId>579</ItemId
   <ItemName>Clasp</ItemName
   <Quantity>1</Quantity>
 </Item>
```

```
                    </Order>
```

Server Access to Order Elements

The preceding example showed uses of get and set methods in a client environment. You can also call those methods in SQL statements in the server:

```
select order_doc>>getOrderElement("CustomerId"),
   order_doc>>getOrderElement("CustomerName"),
   order_doc>>getOrderElement("Date")
 from order_docs

select order_doc>>getItemElement(1, "ItemId"),
   order_doc>>getItemElement(1, "ItemName"),
   order_doc>>getItemElement(1, "Quantity"),
   order_doc>>getItemAttribute(1, "Quantity", "unit")
 from order_docs

update order_docs
 set order_doc = order_doc>>setItemElement(1, "ItemName",
   "Wrench")

update order_docs
 set order_doc = order_doc>>setItemElement(2, "ItemId", "967")

select order_doc>>getItemElement(1, "ItemName"),
   order_doc>>getItemElement(2, "ItemId")
 from order_docs

update order_docs
 set order_doc = order_doc>>setItemAttribute(2, "Quantity",
   "unit", "6")

select order_doc>>getItemAttribute(2, "Quantity", "unit")
 from order_docs
```

Appending and Deleting Items in the XML Document

The **Order** class provides methods for adding and removing items from the Order document.

You can append a new item to the Order document with the **appendItem( )** method, whose parameters specify *ItemId*, *ItemName*, *Quantity*, and *units* for the new item:

```
update order_docs
 set order_doc = order_doc>>appendItem("864",
 "Bracket", "3","12")
```

appendItem( ) is a void method that modifies the instance. When you invoke such a method in an **update** statement, you reference it as shown, as if it were an Order-valued method that returns the updated item.

You delete an existing item from the Order document using **deleteItem( )**. The **deleteItem( )** parameter specifies the number of the item to be deleted. The numbering begins with zero, so the following command deletes the second item from the specified row.

```
update order_docs
 set order_doc = order_doc>>deleteItem(1)
 where id = “1”
```

Using the Hybrid Storage Technique

In the hybrid storage technique, you store the complete XML document in a SQL column and, at the same time, store elements of that document in separate columns. This technique often balances the advantages and disadvantages of element and document storage.

“Using the Document Storage Technique” on page 105 demonstrates how to store the entire XML Order document in the single column *order_docs.order_doc*. Using document storage, you must reference and access the *CustomerId* element in this way:

```
select order_doc>>getOrderElement(“CustomerID”) from order_docs
 where order_doc>>getOrderElement(“CustomerID”) > “222”
```

To access *CustomerId* more quickly and conveniently than with the method call, but without first decomposing the Order into SQL rows and columns:

1 Add a column to the *order_docs* table for the *customer_id*:

```
alter table order_docs
 add  customer_id   varchar(5) null
```

2 Update that new column with extracted *customerId* values.

```
update order_docs
 set customer_id =
 order_doc>>getOrderElement("CustomerId")
```

3 Now, you can reference *CustomerId* values directly:

```
select customer_id from order_docs where
 customer_id > “222”
```

You can also define an index on the column.

Note This technique does not synchronize the extracted *customer_id* column with the *CustomerId* element of the *order_doc* column if you update either value.

A Customizable Example for Different Result Sets

This section demonstrates how you can store XML documents or the data that they contain in an Adaptive Server database using the **ResultSet** class and its methods for handling result sets. You can customize the **ResultSet** class for your database application.

Contrast the ResultSet document type and the Order document type:

- The Order document type is a simplified example designed for a specific purchase-order application, and its Java methods are designed for a specific set of SQL tables for purchase order data. See "A Simple Example for a Specific Result Set" on page 97.
- The ResultSet document type is designed to accommodate many kinds of SQL result sets, and the Java methods designed for it include parameters to accommodate different kinds of SQL queries.

For this example, you create and work with XML **ResultSet** documents that contain the same data as the XML Order documents.

First, create the *orders* table and its data:

```
create table orders
   (customer_id varchar(5) not null,
   order_date datetime not null,
   item_id varchar(5) not null,
   quantity int not null,
   unit smallint default 1)
insert into orders values ("123", "1999/05/07", "987", 5, 1)
insert into orders values ("123", "1999/05/07", "654", 3, 12)
insert into orders values ("123", "1999/05/07", "579", 1, 1)
```

Also, create the following SQL table to store complete XML ResultSet documents:

```
create table resultset_docs
 (id char(5),
 rs_doc jcs.xml.resultsets.ResultSetXml)
```

The ResultSet Document Type

ResultSet documents consist of ResultSetMetaData followed by ResultSetData as shown in the following general form:

```
<?xml version="1.0"?>
 <!DOCTYPE ResultSet SYSTEM 'ResultSet.dtd'>
 <ResultSet>
<ResultSetMetaData>
…
</ResultSetMetaData>
<ResultSetData>
…
</ResultSetData>
</ResultSet>
```

The **ResultSetMetaData** portion of an XML **ResultSet** consists of the SQL metadata returned by the methods of the JDBC **ResultSet** class. The **ResultSetMetaData** for the example result set is:

```
<ResultSetMetaData
 getColumnCount="7">
    <ColumnMetaData
        getColumnDisplaySize="25"
        getColumnLabel="Date"
        getColumnName="Date"
        getColumnType="93"
        getPrecision="0"
        getScale="0"
        isAutoIncrement="false"
        isCurrency="false"
        isDefinitelyWritable="false"
        isNullable="false"
        isSigned="false" />
<ColumnMetaData
        getColumnDisplaySize="5"
        getColumnLabel="CustomerId"
        getColumnName="CustomerId"
        getColumnType="12"
        getPrecision="0"
        getScale="0"
        isAutoIncrement="false"
        isCurrency="false"
        isDefinitelyWritable="false"
        isNullable="false"
        isSigned="false" />
    <ColumnMetaData
        getColumnDisplaySize="50"
        getColumnLabel="CustomerName"
        getColumnName="CustomerName"
        getColumnType="12"
        getPrecision="0"
```

```
            getScale="0"
            isAutoIncrement="false"
            isCurrency="false"
            isDefinitelyWritable="false"
            isNullable="false"
            isSigned="false" />
        <ColumnMetaData
            getColumnDisplaySize="5"
            getColumnLabel="ItemId"
            getColumnName="ItemId"
            getColumnType="12"
            getPrecision="0"
            getScale="0"
            isAutoIncrement="false"
            isCurrency="false"
            isDefinitelyWritable="false"
            isNullable="false"
            isSigned="false" />
        <ColumnMetaData
            getColumnDisplaySize="20"
            getColumnLabel="ItemName"
            getColumnName="ItemName"
            getColumnType="12"
            getPrecision="0"
            getScale="0"
            isAutoIncrement="false"
            isCurrency="false"
            isDefinitelyWritable="false"
            isNullable="false"
            isSigned="false" />
        <ColumnMetaData
            getColumnDisplaySize="11"
            getColumnLabel="Quantity"
            getColumnName="Quantity"
            getColumnType="4"
            getPrecision="0"
            getScale="0"
            isAutoIncrement="false"
            isCurrency="false"
            isDefinitelyWritable="false"
            isNullable="false"
            isSigned="true" />
        <ColumnMetaData
            getColumnDisplaySize="6"
            getColumnLabel="unit"
            getColumnName="unit"
```

```
                                getColumnType="5"
                                getPrecision="0"
                                getScale="0"
                                isAutoIncrement="false"
                                isCurrency="false"
                                isDefinitelyWritable="false"
                                isNullable="false"
                               isSigned="true" />
                    </ResultSetMetaData>
```

The names of the attributes of **ColumnMetaData** are simply the names of the methods of the JDBC **ResultSetMetaData** class, and the values of those attributes are the values returned by those methods.

The **ResultSetData** portion of an XML ResultSet document is a list of *Row* elements, each having a list of *Column* elements. The text value of a *Column* element is the value returned by the JDBC **getString( )** method for the column. The **ResultSetData** for the example is:

```
<ResultSetData>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">987</Column>
       <Column name="ItemName">Coupler</Column>
       <Column name="Quantity">5</Column>
       <Column name="unit">1</Column>
     </Row>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">654</Column>
       <Column name="ItemName">Connecter</Column>
       <Column name="Quantity">3</Column>
       <Column name="unit">12</Column>
     </Row>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">579</Column>
       <Column name="ItemName">Clasp</Column>
       <Column name="Quantity">1</Column>
       <Column name="unit">1</Column>
     </Row>
```

```
   </ResultSetData>
 </ResultSet>
```

The XML DTD for the ResultSetXml Document Type

The DTD for the XML ResultSet document type is:

```
<!ELEMENT ResultSet (ResultSetMetaData ,
      ResultSetData)>
 <!ELEMENT ResultSetMetaData (ColumnMetaData)+>
   <!ATTLIST ResultSetMetaData getColumnCount CDATA
      #IMPLIED>
 <!ELEMENT ColumnMetaData  EMPTY>
 <!ATTLIST ColumnMetaData
   getCatalogName CDATA #IMPLIED
   getColumnDisplaySize CDATA #IMPLIED
   getColumnLabel CDATA #IMPLIED
   getColumnName CDATA #IMPLIED
   getColumnType CDATA #REQUIRED
   getColumnTypeName CDATA #IMPLIED
   getPrecision CDATA #IMPLIED
   getScale CDATA #IMPLIED
   getSchemaName CDATA #IMPLIED
   getTablename CDATA #IMPLIED
   isAutoIncrement (true|false) #IMPLIED
   isCaseSensitive (true|false) #IMPLIED
   isCurrency (true|false) #IMPLIED
   isDefinitelyWritable (true|false) #IMPLIED
   isNullable (true|false) #IMPLIED
   isReadOnly (true|false) #IMPLIED
   isSearchable (true|false) #IMPLIED
   isSigned (true|false) #IMPLIED
   isWritable (true|false) #IMPLIED
   >

<!ELEMENT ResultSetData (Row)*>
<!ELEMENT Row (Column)+>
<!ELEMENT Column (#PCDATA)>
<!ATTLIST Column
   null (true | false) "false"
   name CDATA #IMPLIED
```

The *ResultSetXml* Class for Result Set Documents

This section describes the **ResultSetXml** class that supports the ResultSet DTD.

The **ResultSetXml** class is similar to the **OrderXml** class. It is a subclass of the **JXml** class, which validates a document with the XML ResultSet DTD, and also provides methods for accessing and updating the elements of the contained XML ResultSet document.

- **Constructor: ResultSetXml(String)**

  Validates that the argument contains a valid XML ResultSet document and constructs a *ResultSetXml* object containing that document. For example, if *doc* is a Java String variable containing an XML ResultSet document, read from a file:

  ```
  jcs.xml.resultset.ResultSetXml rsx =
     new jcs.xml.resultset.ResultSetXml(doc);
  ```

- **Constructor: ResultSetXml(query, cdataColumns, colNames, dtdOption, server)**

  The parameters are all String.

  The query parameter is any SQL query that returns a result set.

  The server parameter identifies the Adaptive Server on which to execute the query.

  - If you invoke the method in a client environment, specify the server name.
  - If you invoke the method in a Adaptive Server (in a SQL statement or **isql**), specify either an empty string or the string "jdbc:default:connection," indicating that the query should be executed on the current Adaptive Server.

  The method connects to the server, executes the query, retrieves the SQL result set, and constructs a *ResultSetXml* object with that result set.

  The *cdataColumns* parameter indicates which columns should be XML CDATA sections. The *colNames* parameter indicates whether the resulting XML should specify "name" attributes in the "Column" elements. The dtdOption indicates whether the resulting XML should include the XML DTD for the ResultSet document type in-line, or reference it externally.

  For example:

```
jcs.xml.resultset.ResultSetXml  rsx =
 new jcs.xml.resultset.ResultSetXml
 ("select 1 as 'a', 2 as 'b', 3 ", "none", "yes",
 "external", "antibes:4000?user=sa");
```

This constructor call connects to the server specified in the last argument, evaluates the SQL query given in the first argument, and returns an XML ResultSet containing the data from the result set of the query. This simple SQL query does not reference a table. If the constructor is called in the Adaptive Server, then the server parameter should be an empty string or *jdbc:default:connection*, to indicate a connection to the current server.

- **String toSqlScript(resultTableName, columnPrefix, writeOption, goOption)**

  The parameters are all String.

  The method returns a SQL script with a **create** statement and a list of **insert** statements that re-create the result set data.

  The *resultTableName* parameter is the table name for the **create** and **insert** statements. (SQL result sets do not specify a table name because they could be derived from joins or unions.) The *columnPrefix* parameter is the prefix to use in generated column names, which are needed for unnamed columns in the result set. The *writeOption* parameter indicates whether the script is to include the **create** statement, the **insert** statements, or both. The *goOption* parameter indicates whether the script is to include the **go** commands, which are required in **isql** and not supported in JDBC.

  For example, if *rsx* is a Java variable of type ResultSetXml:

```
rsx>>toSqlScript(“systypes_copy”, “column_”, “both”, “yes”)
```

- **String getColumn(int rowNumber, int columnNumber)**

  rowNumber is the index of a row in the result set; *columnNumber* is the index of a column of the result set. The method returns the text of the specified column.

  For example, if *rsx* is a Java variable of type ResultSetXml:

```
select rsx>>getColumn(3, 4)
```

- **String getColumn(int rowNumber, String columnName)**

  rowNumber is the index of a row in the result set; *columnName* is the name of a column of the result set. The method returns the text of the specified column.

  For example, if *rsx* is a Java variable of type ResultSetXml:

```
select rsx>>getColumn(3, “name”)
```

- **void setColumn(int rowNumber, int columnNumber, newValue)**

rowNumber and *columnNumber* are as described for **getColumn( )**. The method sets the text of the specified column to *newValue*.

For example, if *rsx* is a Java variable of type ResultSetXml:

```
select rsx = rsx>>setColumn(3, 4, “new value”)
```

- **void setColumn(int rowNumber, String columnName, newValue)**

  rowNumber and *columnName* are as described for **getColumn( )**. The method sets the text of the specified column to *newValue*.

  For example, if *rsx* is a Java variable of type ResultSetXml:

  ```
  select rsx = rsx>>setColumn(3, “name”, “new value”)
  ```

- **Boolean allString(int columnNumber, String compOp, String comparand)**

  columnNumber is the index of a column of the result set. *compOp* is a SQL comparison operator (<, >, =, !=, <=, >=). *comparand* is a comparison value. The method returns a value indicating whether the specified comparison is true for all rows of the result set.

  For example, if *rsx* is a Java variable of type ResultSetXml:

  ```
  if rsx>>allString(3, “<”, “compare value”)…
  ```

  This condition is true if in the result set represented by *rsx*, for all rows the value of column 3, is less than “compare value.” This is a String comparison. Similar methods could be used for other data types.

- **Boolean someString(int columnNumber, String compOp, String comparand)**

  columnNumber is the index of a column of the result set. *compOp* is a SQL comparison operator (<, >, =, !=, <=, >=). *comparand* is a comparison value. The method returns aa value indicating whether the specified comparison is true for some row of the result set.

  For example, if *rsx* is a Java variable of type ResultSetXml:

  ```
  if rsx>>someString(3, “<”, “compare value”) …
  ```

  This condition is true if in the result set represented by *rsx*, for some row the value of column 3, is less than “compare value.”

Using the Element Storage Technique

This section uses the *orders* table to illustrate mapping between SQL data and XML ResultSet documents.

- In "Composing a ResultSet XML Document from the SQL Data" on page 120, we generate an XML ResultSet document from the SQL data. We assume that we are the *originator* of the XML *ResultSet* document. We used the resulting XML ResultSet document to describe the ResultSet DTD.
- In "Decomposing the XML ResultSet to SQL Data" on page 121, we re-generate SQL data from the XML ResultSet document. We assume we are the *recipient* of the XML ResultSet document.

Composing a ResultSet XML Document from the SQL Data

You can use Java methods to evaluate a given query and generate an XML result set with the query's data. This example uses a constructor method of the **ResultSetXml** class. For example:

```
new jcs.xml.resultset.ResultSetXml
 ("select 1 as 'a', 2 as 'b', 3 ", "none",
 "yes", "external", "antibes:4000?user=sa");
```

The method uses internal JDBC operations to execute the argument query, and then constructs the XML ResultSet for the query's data.

We can invoke this constructor in a client or in the Adaptive Server:

- If you invoke the constructor in a client, specify a server parameter that identifies the Adaptive Server to be used when evaluating the query. The query is evaluated in the Adaptive Server, but the XML document is assembled in the client.
- If you invoke the constructor in the Adaptive Server, specify a null value or *jdbc:default:connection* for the server. The query is evaluated in the current server and the XML document is assembled there.

Generating a ResultSet in the Client

The **main( )** method of the **OrderResultSetClient** class is invoked in a client environment. **main( )** invokes the constructor of the **ResultSetXml** class to generate an XML ResultSet. The constructor executes the query, retrieves its metadata and data using JDBC **ResultSet** methods, and assembles an XML ResultSet document with the data.

```
import java.io.*;
import jcs.util.*;
public class OrderResultSetClient {
    public static void main (String[] args) {
       try{
       String orderQuery = "select order_date as Date,
              c.customer_id as CustomerId, "
          + "customer_name as CustomerName, "
          + "o.item_id as ItemId, i.item_name as ItemName, "
          + "quantity as Quantity, o.unit as unit "
          + "from customers c, orders o, items i "
          + "where c.customer_id=o.customer_id and
             o.item_id=i.item_id " ;
       jcs.xml.resultset.ResultSetXml rsx
              = new jcs.xml.resultset.ResultSetXml(orderQuery,
              "none", "yes", "external",
              "antibes:4000?user=sa");
          FileUtil.string2File("OrderResultSet.xml",
             rsx.getXmlText());
   } catch (Exception e) {
          System.out.println("Exception:");
          e.printStackTrace();
       }
    }
 }
```

Generating a ResultSet in Adaptive Server

The following SQL script invokes the constructor of the **ResultSetXml** class in a server environment:

```
declare @rsx jcs.xml.resultset.ResultSetXml
select @rsx = new jcs.xml.resultset.ResultSetXml
   (“select 1 as ‘a’, 2 as ‘b’, 3 ”, “none”, “yes”, “external”, “”);
insert into resultset_docs values (“1”, @rsx)
```

Decomposing the XML ResultSet to SQL Data

In this section, you decompose an existing ResultSet document to SQL data.

- In section “Decomposing Data from an XML Order into SQL” on page 103, you invoke the **order2Sql( )** method of the **OrderXml** class to decompose an XML Order document into SQL data. **order2Sql( )** directly inserts the extracted data into a SQL table.

- In this example, the to **toSqlScript( )** method of the **ResultSetXml** class decomposes an XML ResultSet document into SQL data. Instead of directly inserting extracted data into a SQL table, however, **toSqlScript( )** returns a SQL script with generated **insert** statements.

The two approaches are equivalent.

Decomposing the XML ResultSet Document in the Client

The **main( )** method of **ResultSetXml** is executed in a client environment. It copies the file *OrderResultSet.xml*, constructs a *ResultSetXml* object containing the contents of that file, and invokes the **toSqlScript( )** method of that object to generate a SQL script that recreates the data of the result set. The method stores the SQL script in the file *order-resultset-copy.sql*.

```
import java.io.*;
import jcs.util.*;
public class ResultSet2Sql{
    public static void main (String[] args) {
       try{
          String xml = FileUtil.file2String("OrderResultSet.xml");
          jcs.xml.resultset.ResultSetXml rsx
          = new jcs.xml.resultset.ResultSetXml(xml);
          String sqlScript
          = rsx.toSqlScript("orderresultset_copy", "col_",
              "both", "no");
          FileUtil.string2File("order-resultset-copy.sql",
          sqlScript);
          jcs.util.ExecSql.statement(sqlScript,
          “antibes:4000?user=sa”);
       } catch (Exception e) {
          System.out.println("Exception:");
          e.printStackTrace();
       }
    }
 }
```

The following is the SQL script generated by **ResultSet2Sql**.

```
set quoted_identifier on
 create table orderresultset_copy (
    Date datetime not null ,
    CustomerId varchar (5) not null ,
    CustomerName varchar (50) not null ,
    ItemId varchar (5) not null ,
    ItemName varchar (20) not null ,
    Quantity integer not null ,
```

```
    unit smallint not null
 )
insert into orderresultset_copy values (
    '1999-07-04 00:00:00.0',   '123',
 'Acme Alpha',    '987', 'Widget', 5,  1 )
 insert into orderresultset_copy values (
    '1999-07-04 00:00:00.0',  '123',
 'Acme Alpha',  '654',
 'Medium connecter',  3,  12 )
 insert into orderresultset_copy values (
 '1999-07-04 00:00:00.0',  '123',
 'Acme Alpha',  '579',  'Type 3 clasp',  1,  1 )
```

The SQL script includes the **set quoted_identifier on** command for those cases where the generated SQL uses quoted identifiers.

Decomposing the XML ResultSet Document in Adaptive Server

The following SQL script invokes the **toSqlScript( )** method in Adaptive Server and then creates and populates a table with a copy of the result set data.

```
declare @rsx jcs.xml.resultset.ResultSetXml
select @rsx = rs_doc from resultset_docs where id=1
select @script = @rsx>>toSqlScript("resultset_copy",
"column_", "both", "no")
declare @I integer
select @I = jcs.util.ExecSql.statement(@script, "")
```

Using the Document Storage Technique

This section shows examples of storing XML ResultSet documents in single SQL columns and techniques for referencing and updating the column elements.

Storing an XML ResultSet Document in a SQL Column

The following SQL script generates an XML ResultSet document and stores it in a table:

```
declare @query java.lang.StringBuffer
select @query = new java.lang.StringBuffer()
 -- The following "appends" build up a SQL select statement in
    the @query variable
 -- We use a StringBuffer, and the append method, so that the
    @query can be as long as needed.
```

```
select @query>>append("select order_date as Date,
   c.customer_id as CustomerId, ")
select @query>>append("customer_name as CustomerName, ")
select @query>>append("o.item_id as ItemId, i.item_name as
   ItemName, ")
select @query>>append("quantity as Quantity, o.unit as unit " )
select @query>>append("from customers c, orders o, items i ")
select @query>>append("where c.customer_id=o.customer_id and
   o.item_id=i.item_id  ")
declare @rsx jcs.xml.resultset.ResultSetXml
select  @rsx = new jcs.xml.resultset.ResultSetXml
   (@query>>toString(), 'none', 'yes',  'external' , '')
insert into resultset_docs values("1",  @rsx)
```

Accessing the Columns of Stored ResultSet Documents

In "Storing an XML ResultSet Document in a SQL Column" on page 123 you inserted a complete XML ResultSet document into the *rs_doc* column of the *resultset_docs* table. In this section, use methods of the **ResultSetXml** class to reference and update a stored ResultSet.

A Client-Side Call

The **main( )** method of the **ResultSetElements** class is executed in a client environment. It copies the file *OrderResultSet.xml*, constructs a ResultSetXml document from it, and then accesses and updates the columns of the ResultSet.

```
import java.io.*;
import jcs.util.*;
public class ResultSetElements {
    public static void main ( String[] args) {
       try{
       String xml =
          FileUtil.file2String("OrderResultSet.xml");
       jcs.xml.resultset.ResultSetXml rsx
                 = new jcs.xml.resultset.ResultSetXml(xml);
       // Get the columns containing customer and date info
       String cname = rsx.getColumn(0, "CustomerName");
       String cid   = rsx.getColumn(0, "CustomerId");
       String date = rsx.getColumn(0, "Date");
       // Get the elements for item 1 (numbering from 0)
       String iName1 = rsx.getColumn(1, "ItemName");
       String iId1 = rsx.getColumn(1, "ItemId");
       String iQ1 = rsx.getColumn(1, "Quantity");
       String iU = rsx.getColumn(1,  "unit");
       System.out.println("\nBEFORE UPDATE: ");
```

```
      System.out.println("\n   "+date+ "  "+ cname + " " +
         cid);
      System.out.println("\n   "+ iName1+" "+iId1+" "
         + iQ1 + " " + iU + "\n");
      //  Set the elements for item 1 (numbering from 0)
      rsx.setColumn(1, "ItemName", "Flange");
      rsx.setColumn(1, "ItemId", "777");
      rsx.setColumn(1, "Quantity","3");
      rsx.setColumn(1,  "unit", "13");
      //  Get the updated elements for item 1 (numbering
         from 0) iName1 = rsx.getColumn(1, "ItemName");
      iId1 = rsx.getColumn(1, "ItemId");
      iQ1 = rsx.getColumn(1, "Quantity");
      iU = rsx.getColumn(1,  "unit");
      System.out.println("\nAFTER UPDATE: ");
      System.out.println("\n   "+date+ "  "+ cname + " " +
         cid);
      System.out.println("\n   "+ iName1+" "+iId1+" "
         + iQ1 + " " + iU + "\n");

         //  Copy the updated document to another file
      FileUtil.string2File("Order-updated.xml",
         rsx.getXmlText());
      } catch (Exception e) {
      System.out.println("Exception:");
      e.printStackTrace();
      }
   }
}
```

The **FileUtil.string2File( )** method stores the updated ResultSet in the file *Order-updated.xml*. The ResultSetMetaData of the updated document is unchanged. The updated ResultSetData of the document is as follows with new values in the second item.

```
<ResultSetData>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
       <Column name="CustomerId">123</Column>
       <Column name="CustomerName">Acme Alpha</Column>
       <Column name="ItemId">987</Column>
       <Column name="ItemName">Widget</Column>
       <Column name="Quantity">5</Column>
       <Column name="unit">1</Column>
     </Row>
     <Row>
       <Column name="Date">1999-07-04 00:00:00.0</Column>
```

```
      <Column name="CustomerId">123</Column>
      <Column name="CustomerName">Acme Alpha</Column>
      <Column name="ItemId">777</Column>
      <Column name="ItemName">Flange</Column>
      <Column name="Quantity">3</Column>
      <Column name="unit">13</Column>
    </Row>
    <Row>
      <Column name="Date">1999-07-04 00:00:00.0</Column>
      <Column name="CustomerId">123</Column>
      <Column name="CustomerName">Acme Alpha</Column>
      <Column name="ItemId">579</Column>
      <Column name="ItemName">Type 3 clasp</Column>
      <Column name="Quantity">1</Column>
      <Column name="unit">1</Column>
    </Row>
  </ResultSetData>
</ResultSet>
```

A Server-Side Script

Using the SQL script in "Storing XML Order Documents in SQL Columns" on page 105, you stored complete XML ResultSet documents in the *rs_doc* column of the *resultset_docs* table. The following SQL commands, executed in a server environment, reference and update the columns contained in those documents.

You can select columns by name or by number:

- Select the columns of row 1, specifying columns by name:

```
select rs_doc>>getColumn(1, "Date"),
 rs_doc>>getColumn(1, "CustomerId"),
 rs_doc>>getColumn(1, "CustomerName"),
 rs_doc>>getColumn(1, "ItemId"),
 rs_doc>>getColumn(1, "ItemName"),
 rs_doc>>getColumn(1, "Quantity"),
 rs_doc>>getColumn(1,  "unit")
  from resultset_docs
```

- Select the columns of row 1, specifying columns by number:

```
select rs_doc>>getColumn(1, 0),
 rs_doc>>getColumn(1, 1),
 rs_doc>>getColumn(1, 2),
 rs_doc>>getColumn(1, 3),
 rs_doc>>getColumn(1, 4),
 rs_doc>>getColumn(1, 5),
```

```
rs_doc>>getColumn(1,  6)
from resultset_docs
```

Specify some non-existing columns and rows. Those references return null values.

```
Select rs_doc>>getcolumn(1, "itemid"),
 rs_doc>>getcolumn(1, "xxx"),
 rs_doc>>getcolumn(1, "Quantity"),
 rs_doc>>getcolumn(99,  "unit"),
 rs_doc>>getColumn(1, 876)
 from resultset_docs
```

Update columns in the stored ResultSet document:

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(1, "ItemName",
"Wrench")
 where id=”1”
update resultset_docs
 set rs_doc = rs_doc>>setColumn(1, "ItemId", "967")
 where id=”1”
update resultset_docs
 set rs_doc = rs_doc>>setColumn(1,  "unit", "6")
 where id=”1”
select rs_doc>>getColumn(1, "ItemName"),
 rs_doc>>getColumn(1, "ItemId"),
 rs_doc>>getColumn(1,  "unit")
 from resultset_docs
 where id=”1”
```

Quantified Comparisons in Stored ResultSet Documents

ResultSetXml contains two methods, **allString( )** and **someString( )**, for quantified searches on columns of a ResultSetXML document. To illustrate these two methods, first create some example rows in the *order_results* table.

The *order_results* table has been initialized with one row, whose *id* = “1” and whose *rs_doc* column contains the original Order used in all examples.

The following statements copy that row twice, assigning *id* values of “2” and “3” to the new rows. The *order_results* table now has three rows, with *id* column values of “1,” “2,” and “3” and the original Order.

```
insert into resultset_docs(id, rs_doc) select "2", rs_doc
 from resultset_docs where id="1"
insert into resultset_docs (id, rs_doc) select "3", rs_doc
 from resultset_docs where id="1"
```

The following statements modify the row with an *id* column value of “1” so that all three items have an *ItemId* of “100”:

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(0, "ItemId", "100")
 where id="1"
update resultset_docs
 set rs_doc = rs_doc>>setColumn(1, "ItemId", "110")
 where id="1"
update resultset_docs
 set rs_doc = rs_doc>>setColumn(2, "ItemId", "120")
 where id="1"
```

The following **update** statement modifies the row with *id* = “3” so that its second item (from 0) has an *ItemId* of “999”:

```
update resultset_docs
 set rs_doc = rs_doc>>setColumn(2, "ItemId", "999")
 where id="3"
```

The following **select** statement displays the *id* column and the three *ItemId* values for each row:

```
select id, rs_doc>>getColumn(0, "ItemId"),
 rs_doc>>getColumn(1, "ItemId"),
 rs_doc>>getColumn(2, "ItemId")
 from resultset_docs
```

The results of the **select** are:

```
1     100      110        120
2     987      654        579
3     987      654        999
```

Note the following:

- The row with *id* of “2” is the original Order data.
- The row with *id* of “1” has been modified so that *all ItemId*s for that row are less than “200.”
- The row with *id* of “3” has been modified so that *some ItemId* for that row is greater than or equal to “9999,”

The following expresses the these quantifications with the **allString( )** and **someString( )** methods:

```
select id, rs_doc>>allString(3, "<", "200") as “ALL test”
 from resultset_docs
select id, rs_doc>>someString(3, ">=", "999") as “SOME test”
 from resultset_docs
```

```
select id as “id for ALL test” from resultset_docs
 where rs_doc>>allString(3, "<", "200")>>booleanValue() = 1
select id as “id for SOME test” from resultset_docs
 where rs_doc>>someString(3, ">=", "999")>>booleanValue() = 1
```

The first two statements show the quantifier in the **select** list and give these results:

Id	“all” test	“some” test
1	true	false
2	false	false
3	false	true

The last two statements show the quantifier in the **where** clause and give these results:

- Id for “all” test = “3”
- Id for “some” test = “1”

In the examples with the quantifier method in the **where** clause, note that:

- The **where** clause examples compare the method results with an integer value of 1. SQL does not support the Boolean datatype as a function value, but instead treats Boolean as equivalent to integer values 1and 0, for true and false.
- The **where** clause examples use the **booleanValue( )** method. The **allString( )** and **someString( )** methods return type java.lang.Boolean, which is not compatible with SQL integer. The Java **booleanValue( )** method returns the simple Boolean value from the Boolean object, which is compatible with SQL integer. This behavior is a result of merging the SQL and Java type systems.

The quantifier methods return java.lang.Boolean instead of simply Java boolean so that they can return null when the column is out of range, which is consistent with the SQL treatment of out-of-range conditions.

The following statements show quantifier references that specify column 33, which does not exist in the data:

```
select id, rs_doc>>allString(33, "<", "200") as “ALL test”
 from resultset_docs
select id as “id for ALL test” from resultset_docs
 where rs_doc>>allString(33, "<", "200")>>booleanValue() = 1
```

Id	“all” test
1	NULL
2	NULL
3	NULL

The id for the “all” test = (empty).

Using the Hybrid Storage Technique

For faster and easier access to the *CustomerID* element, add a new *customer_id* column to the *resultset_docs* table, and populate it with extracted *CustomerID* elements:

```
alter table resultset_docs
   add  customer_id   varchar(5) null
update resultset_docs
   set customer_id = rs_doc>>getColumn(1, "CustomerId")
```

XML ResultSet Documents: Invalid XML Characters

This section describes two techniques for dealing with XML markup characters in the result set.

When data values contain XML markup characters, you can enclose these values in a CDATA section.

- When column names are quoted identifiers that contain XML markup characters, you can substitute the quotes and markup characters with CML entity symbols.

Each technique is described in the following sections.

Using CDATA Sections

The *cdata* parameter of the **ResultSetXml** constructor indicates which (if any) columns of the SQL result set contain character data to be bracketed as CDATA sections in the output XML. The *cdata* parameter can be “all,” “none,” or a string of zero or one characters, where a “1” in the I-th position indicates that the I-th column should be bracketed as a CDATA section.

For example, create the table *cdata* in which data values in columns 2, 3, and 4 contain XML markup characters that must be bracketed as CDATA section in the output:

```
create table cdata (
 id int,
 a varchar(250),
 b varchar(250),
 c varchar(250)
    )
 go
insert into cdata values (
 1,
 "<p>some samples:</p><ol><li>first</li>
   <li>second</li></ol>",
 "x > y || w  & z",
 "x > y || w & z"
 )
```

The following SQL statement generates an XML ResultSet document for this table, specifying a value “0111” for the *cdata* parameter.

```
insert into resultsets (id, rs)
   values ("2", new jcs.xml.resultset.ResultSetXml(
      "select * from cdata", ’0111’, ’yes’, ’external’, ’’))
```

This SQL statement generates a SQL script for that XML ResultSet:

```
update resultsets
 set script =
   rs>>toSqlScript("markup_col_names",
   "col_", "both", "yes")
 where id="2"
```

The following utility calls retrieve the XML ResultSet and its SQL script:

```
java jcs.util.FileUtil -S "$SERVER" -A getstring -O cdata.xml \
   -Q "select new jcs.util.StringWrap(rs>>getXmlText()) from
      resultsets where id=’2’"
java jcs.util.FileUtil -S "$SERVER" -A getstring -O cdata.script\
   -Q "select new jcs.util.StringWrap(script) from resultsets
      where id=’2’"
```

This is the XML ResultSet:

```
<?xml version="1.0"?>
<!DOCTYPE ResultSet SYSTEM ’ResultSet.dtd’>
<ResultSet>
   <ResultSetMetaData getColumnCount="4">
     <ColumnMetaData getColumnDisplaySize="11" getColumnLabel="id"
```

```
getColumnName="id" getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false" isDefinitelyWritable="false"
isNullable="false" isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="250" getColumnLabel="a"
getColumnName="a" getColumnType="12" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false" isDefinitelyWritable="false"
isNullable="false" isSigned="false" />
     <ColumnMetaData getColumnDisplaySize="250" getColumnLabel="b"
getColumnName="b" getColumnType="12" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false" isDefinitelyWritable="false"
isNullable="false" isSigned="false" />
     <ColumnMetaData getColumnDisplaySize="250" getColumnLabel="c"
getColumnName="c" getColumnType="12" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false" isDefinitelyWritable="false"
isNullable="false" isSigned="false" />
   </ResultSetMetaData>
   <ResultSetData>
     <Row>
       <Column name="id">1</Column>
       <Column name="a">
       <![CDATA[<p>some samples:
          </p><ol><li>first</li><li>second</li></ol>]]>
       </Column>
       <Column name="b">
       <![CDATA[x > y || w  & z]]>
       </Column>
       <Column name="c">
       <![CDATA[x > y || w & z]]>
       </Column>
     </Row>
   </ResultSetData>
 </ResultSet>
```

This is the SQL script:

```
set quoted_identifier on
 create table markup_col_names (
    id integer not null ,
    a varchar (250) not null ,
    b varchar (250) not null ,
    c varchar (250) not null
 )
 insert into markup_col_names values (
    1,
    '<p>some samples:</p><ol><li>first</li><li>second</li></ol>',
    'x > y || w  & z',
    'x > y || w & z'
```

```
)
```

Column Names

The XML generated for a SQL result set specifies the column names of the result set in the ResultSetMetaData section and in the ResultSetData section.

The following SQL **select** specifies a result set:

```
select 1 as “A>2”, 2 as “B  & 3”, 3 as “A<<b”, 4 as
 ”D ””or”” e”
```

The result set has a single row, whose column values are 1, 2, 3, and 4. The column names of those columns are quoted identifiers that contain XML markup characters.

Since the ResultSetXml document for such a result set specifies the column names in XML attributes, the quotation marks and XML markup characters in those names must be replaced with XML entity symbols.

This problem cannot be handled with CDATA sections, since you cannot use CDATA sections in attribute values.

The following is a SQL script that generates the ResultSetXml document for the result set, then generates the SQL script for that ResultSetXml document.

Store the generated *ResultSetXml* document in the following table:

```
create table resultsets
 (id char(5) unique,
 rs jcs.xml.resultsets.ResultSetXml null,
 script java.lang.String null)
```

The following SQL statement generates the XML ResultSet document and stores it into the *resultsets* table:

```
insert into resultsets (id, rs)
    values ("1", new jcs.xml.resultsets.ResultSetXml(
        "select 1 as ""A > 2"", 2 as ""b  & 3"",
              3 as ""a<<b"", 4 as ""d """"or"""" e"" ",
                'none', 'yes', 'external', '' ))
```

This SQL statement generates the SQL script for the XML ResultSet:

```
update resultsets
   set script = rs>>toSqlScript("markup_col_names", "col_",
   "create", "yes")
   where id="1"
```

The following utility calls retrieve the XML ResultSet and its SQL script into client files *cdata.xml* and *cdata.script*.

```
java jcs.util.FileUtil -S "$SERVER" -A getstring -O cdata.xml \
   -Q "select new jcs.util.StringWrap(rs>>getXmlText()) from
      resultsets where id='2'"
java jcs.util.FileUtil -S "$SERVER" -A getstring -O cdata.script\
   -Q "select new jcs.util.StringWrap(script) from resultsets
      where id='2'"
```

The XML ResultSet document for the CDATA example is:

```
<?xml version="1.0"?>
<!DOCTYPE ResultSet SYSTEM 'ResultSet.dtd'>
<ResultSet>
   <ResultSetMetaData getColumnCount="4">
     <ColumnMetaData getColumnDisplaySize="11"
getColumnLabel="A > 2" getColumnName="A > 2"
getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false"
isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="11"
getColumnLabel="b & 3" getColumnName="b & 3"
getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false"
isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="11"
getColumnLabel="a<<b" getColumnName="a<<b"
getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false"
isSigned="true" />
     <ColumnMetaData getColumnDisplaySize="11"
getColumnLabel="d 'or' e" getColumnName="d 'or' e"
getColumnType="4" getPrecision="0" getScale="0"
isAutoIncrement="false" isCurrency="false"
isDefinitelyWritable="false" isNullable="false"
isSigned="true" />
   </ResultSetMetaData>
   <ResultSetData>
     <Row>
       <Column name="A > 2">1</Column>
       <Column name="b & 3">2</Column>
       <Column name="a<<b">3</Column>
       <Column name="d 'or' e">4</Column>
```

```
      </Row>
    </ResultSetData>
  </ResultSet>
```

The following is the output SQL script for the CDATA example:

```
set quoted_identifier on
 create table markup_col_names (
    "A > 2" integer not null ,
    "b  & 3" integer not null ,
    "a<<b" integer not null ,
    "d ""or"" e" integer not null
```

CHAPTER 6

Debugging Java in the Database

This chapter describes the Sybase Java debugger and how you can use it when developing Java in Adaptive Server.

These topics are discussed:

Introduction to Debugging Java

You can use the Sybase Java debugger to test Java classes and fix problems with them.

How the Debugger Works

The Sybase Java debugger is a Java application that runs on a client machine. It connects to the database using the Sybase jConnect JDBC driver.

The debugger debugs classes running in the database. You can step through the source code for the files as long as you have the Java source code on the disk of your client machine. (Remember, the compiled classes are installed in the database, but the source code is not).

Requirements for Using the Java Debugger

To use the Java debugger, you need:

- A Java runtime environment such as the Sun Microsystems Java Runtime Environment, or the full Sun Microsystems JDK on your machine.
- The source code for your application on your client machine.

What You Can Do with the Debugger

Using the Sybase Java debugger, you can:

- Trace execution – Step line by line through the code of a class running in the database. You can also look up and down the stack of functions that have been called.
- Set breakpoints – Run the code until you hit a breakpoint, and stop at that point in the code.
- Set break conditions – Breakpoints include lines of code, but you can also specify conditions when the code is to break. For example, you can stop at a line the tenth time it is executed, or only if a variable has a particular value. You can also stop whenever a particular exception is thrown in the Java application.

- Browse classes – You can browse through the classes installed into the database that the server is currently using.
- Inspect and set variables – You can inspect the values of variables alter their value when the execution is stopped at a breakpoint.
- Inspect and break on expressions – You can inspect the value of a wide variety of expressions.

Using the Debugger

This section describes how to use the Java debugger. The next section provides a simple tutorial.

Starting the Debugger and Connecting to the Database

The debugger is the JAR file *Debug.jar*, installed in your Adaptive Server installation directory in *$SYBASE/$SYBASE_ASE/debugger*. If it is not already present, add this file as the first element to your CLASSPATH environment variable.

Debug.jar contains many classes. To start the debugger you invoke the **sybase.vm.Debug** class, which has a **main( )** method.You can start the debugger in three ways:

- Run the *jdebug* script located in *$SYBASE/$SYBASE_ASE/debugger*.

  "A Debugging Tutorial" on page 147 provides a sample debugging session using the *jdebug* script.

- From the command line, enter:

  ```
  java sybase.vm.Debug
  ```

  In the Connect window, enter a URL, user login name, and password to connect to the database.

- From Sybase Central:

  a Start Sybase Central and open the Utilities folder, under Adaptive Server Enterprise.

  b Double-click the Java debugger icon in the right panel.

  c In the Connect window, enter a URL, user login name, and password to connect to the database.

Compiling Classes for Debugging

Java compilers such as the Sun Microsystems **javac** compiler can compile Java classes at different levels of optimization. You can opt to compile Java code so that information used by debuggers is retained in the compiled class files.

If you compile your source code without using switches for debugging, you can still step through code and use breakpoints. However, you cannot inspect the values of local variables.

To compile classes for debugging using the **javac** compiler, use the **-g** option:

```
javac -g ClassName.java
```

Attaching to a Java VM

When you connect to a database from the debugger, the Connection window shows all currently active Java VMs under the user login name. If there are none, the debugger goes into *wait mode*. Wait mode works like this:

- Each time a new Java VM is started, it shows up in the list.
- You may choose either to debug the new Java VM or to wait for another one to appear.
- Once you have passed on a Java VM, you lose your chance to debug that Java VM. If you then decide to attach to the passed Java VM, you must disconnect from the database and reconnect. At this time, the Java VM appears as active, and you can attach to it.

The Source Window

The Source window:

- Displays Java source code, with line numbers and breakpoint indicators (an asterisk in the left column).
- Displays execution status in the status box at the bottom of the window.
- Provides access to other debugger windows from the menu.

The Debugger Windows

The debugger has the these windows:

- Breakpoints window – Displays the list of current breakpoints.
- Calls window – Displays the current call stack.

- Classes window – Displays a list of classes currently loaded in the Java VM. In addition, this window displays a list of methods for the currently selected class and a list of static variables for the currently selected class. In this window you can set breakpoints on entry to a method or when a static variable is written.
- Connection window – The Connection window is shown when the debugger is started. You can display it again if you wish to disconnect from the database.
- Exceptions window – You can set a particular exception on which to break, or choose to break on all exceptions.
- Inspection window – Displays current static variables, and allows you to modify them. You can also inspect the value of a Java expression, such as the following:
  - Local variables
  - Static variables
  - Expressions using the dot operator
  - Expressions using subscripts []
  - Expressions using parentheses, arithmetic, or logical operators.

    For example, the following expressions could be used:

    ```
    x[i].field
    q + 1
    i == 7
    (i + 1)*3
    ```
- Locals window – Displays current local variables, and allows you to modify them.
- Status window – Displays messages describing the execution state of the Java VM.

Options

The complete set of options for stepping through source code are displayed on the Run menu. They include the following:

Function	Shortcut key	Description
Run	F5	Continue running until the next breakpoint, until the Stop item is selected, or until execution finishes.
Step Over	F7 or Space	Step to the next line in the current method. If the line steps into a different method, step over the method, not into it. Also, step over any breakpoints within methods that are stepped over.
Step Into	F8 or i	Step to the next line of code. If the line steps into a different method, step into the method.
Step Out	F11	Complete the current method, and break at the next line of the calling method.
Stop		Break execution.
Run to Selected	F6	Run until the currently selected line is executed and then break.
Home	F4	Select the line where the execution is broken.

Setting Breakpoints

When you set a breakpoint in the debugger, the Java VM stops execution at that breakpoint. Once execution is stopped, you can inspect and modify the values of variables and other expressions to better understand the state of the program. You can then trace through execution step by step to identify problems.

Setting breakpoints in the proper places is a key to efficiently pinpointing the problem execution steps.

The Java debugger allows you to set breakpoints not only on a line of code, but on many other conditions. This section describes how to set breakpoints using different conditions.

Breaking on a Line Number

When you break on a particular line of code, execution stops whenever that line of code is executed.

To set a breakpoint on a particular line:

- In the Source window, select the line and press F9.

  Alternatively, you can double-click a line.

When a breakpoint is set on a line number, the breakpoint is shown in the Source window by an asterisk in the left column. If the Breakpoints window is open, the method and line number is displayed in the list of breakpoints.

You can toggle the breakpoint on and off by repeatedly double-clicking or pressing F9.

Breaking on a Class Method

When you break on a method, the break point is set on the first line of code in the method that contains an executable statement.

To set a breakpoint on a class method:

1 From the Source window, choose Break→ New. The Break At window is displayed.

2 Enter the name of a method in which you wish execution to stop. For example:

```
JDBCExamples.selecter
```

stops execution whenever the **JDBCExamples.selecter( )** method is entered.

When a breakpoint is set on a method, the breakpoint is shown in the Source window by an asterisk in the left column of the line where the breakpoint actually occurs. If the Breakpoints window is open, the method is displayed in the list of breakpoints.

Using Counts with Breakpoints

If you set a breakpoint on a line that is in a loop, or in a method that is frequently invoked, you may find that the line is executed many times before the condition you are really interested in takes place. The debugger allows you to associate a count with a breakpoint, so that execution stops only when the line is executed a set number of times.

To associate a count with a breakpoint:

1 From the Source window, select Break→Display. The Breakpoints window is displayed.

2 In the Breakpoints window, click a breakpoint to select it.

3 Select Break→Count. A window is displayed with a field for entering a number of iterations. Enter an integer value. The execution will stop when the line has been executed the specified number of times.

Using Conditions with Breakpoints

The debugger allows you to associate a condition with a breakpoint, so that execution stops only when the line is executed and the condition is met.

To associate a condition with a breakpoint:

1 From the Source window, select Break→Display. The Breakpoints window is displayed.

2 In the Breakpoints window, click a breakpoint to select it.

3 Select Break→Condition. A window is displayed with a field for entering an expression. The execution will stop when the condition is true.

The expressions used here are the same as those that can be used in the Inspection window, and include the following:

- Local variables
- Static variables
- Expressions using the dot operator
- Expressions using subscripts []
- Expressions using parentheses, arithmetic, or logical operators.

Breaking When Execution Is Not Interrupted

With a single exception, breakpoints can only be set when program execution is interrupted. If you clear all breakpoints, and run the program you are debugging to completion, you can no longer set a breakpoint on a line or at the start of a method. Also, if a program is running in a loop, execution is continuing and is not interrupted.

To debug your program under either of these conditions, select Run→Stop from the Source window. This stops execution at the next line of Java code that is executed. You can then set breakpoints at other points in the code.

Disconnecting from the Database

When the program has run to completion, or at anytime during debugging, you can disconnect from the database from the Connect window. Then, exit the Source window and reconnect to the database after the debug program terminates.

A Debugging Tutorial

This section takes you through a simple debugging session.

Before You Begin

The source code for the class used in this tutorial is located in *$SYBASE/$SYBASE_ASE/sample/JavaSql/manual-examples/JDBCExamples.java*.

Before you run the debugger, compile the source code using the **javac** command with the **-g** option.

See "Creating Java Classes and JARs" on page 22 for complete instructions for compiling and installing Java classes in the database.

Start the Java Debugger and Connect to the Database

You can start the debugger and connect to the database using a script, command line options, or Sybase Central. In this tutorial, we use *jdebug* to start the debugger. You can use any database.

Follow these steps:

1 Start Adaptive Server.

2 If Java queries have not yet been executed on your server, run any Java query to initialize the Java subsystem and start a Java VM.

3 Run the *$SYBASE/$SYBASE_ASE/debugger/jdebug* script. *jdebug* prompts you for these parameters:

   a Machine name of the Adaptive Server

   b Port number for the database

   c Your login name

   d Your password

   e An alternate path to *Debug.jar* if its location is not in your CLASSPATH

Once the connection is established, the debugger window displays a list of available Java VMs or "Waiting for a VM."

Attach to a Java VM

To attach to a Java VM from your user session:

1 With the debugger running, connect to the sample database from **isql** as the **sa**:

```
$SYBASE/bin/isql -Usa -P
```

Note You cannot start Java execution from the debugger. To start a Java VM you must carry out a Java operation from another connection using the same user name.

2 Execute Java code using the following statements:

```
select JDBCExamples.serverMain('createtable')
select JDBCExamples.serverMain('insert')
select JDBCExamples.serverMain('select')
```

The Sybase Java VM starts in order to retrieve the Java objects from the table. The debugger immediately stops execution of the Java code.

The debugger Connection window displays the Java VMs belonging to the user in this format:

VM#: "*login_name*, spid:*spid#*"

3 In the debugger Connection window, click the Java VM you want and then click **Attach to VM**. The debugger attaches to the Java VM and the Source window appears. The Connection window disappears.

Next, enable the Source window to show the source code for the method. The source code is available on disk.

Load Source Code into the Debugger

The debugger looks for source code files. You need to make the *$SYBASE/$SYBASE_ASE/sample/JavaSql/manual-examples/* subdirectory available to the debugger, so that the debugger can find source code for the class currently executing in the database.

To add a source code location to the debugger:

1 From the Source window, select File→Source Path. The Source Path window displays.

2 From the Source Path window, select Path→Add. Enter the following location into the text box:

```
$SYBASE/$SYBASE_ASE/sample/JavaSql/
    manual-examples/
```

The source code for the **JDBCExamples** class displays in the window, with the first line of the Query method **serverMain( )** highlighted. The Java debugger has stopped execution of the code at this point.

You can now close the Source Path window.

Step Through Source Code

You can step through source code in the Java debugger in several ways. In this section we illustrate the different ways you can step through code using the **serverMain( )** method.

When execution pauses at a line until you provide further instructions, we say that the execution **breaks** at the line. The line is a **breakpoint**. Stepping through code is a matter of setting explicit or implicit breakpoints in the code, and executing code to that breakpoint.

Following the previous section, the debugger should have stopped execution of **JDBCExamples.serverMain( )** at the first statement:

Examples

Here are some steps you can try:

1 Stepping into a function – press F7 to step to the next line in the current method.

2 Press F8 to step into the function **doAction( )** in line 99.

3 Run to a selected line. You are now in function **doAction( )**. Click on line 155 and press F6 to run to that line and break:

```
String workString = "Action(" + action + ")";
```

4 Set a breakpoint and execute to it – select line 179 and press F9 to set a breakpoint on that line when running "isql> select JDBCExamples.serverMain('select')":

```
workString + = selecter(con);
```

Press F5 to execute to that line.

5 Experiment – try different methods of stepping through the code. End with F5 to complete the execution.

When you have completed the execution, the Interactive SQL Data window displays:

```
Action(select) – Row with id = 1: name(Joe Smith)
```

Inspecting and Modifying Variables

You can inspect the values of both local variables (declared in a method) and class static variables in the debugger.

Inspecting Local Variables

You can inspect the values of local variables in a method as you step through the code, to better understand what is happening.

To inspect and change the value of a variable:

1 Set a breakpoint at the first line of the **selecter( )** method from the Breakpoint window. This line is:

```
String sql = "select name, home from xmp where
   id=?";
```

2 In Interactive SQL, enter the following statement again to execute the method:

```
select JDBCExamples.serverMain('select')
```

The query executes only as far as the breakpoint.

3 Press F7 to step to the next line. The *sql* variable has now been declared and initialized.

4 From the Source window, select Window→Locals. The Local window appears.

The Locals window shows that there are several local variables. The *sql* variable has a value of zero. All others are listed as not in scope, which means they are not yet initialized.

You must add the variables to the list in the Inspect window.

5 In the Source window, press F7 repeatedly to step through the code. As you do so, the values of the variables appear in the Locals window.

If a local variable is not a simple integer or other quantity, then as soon as it is set a + sign appears next to it. This means the local variable has fields that have values. You can expand a local variable by double-clicking the + sign or setting the cursor on the line and pressing **Enter**.

6 Complete the execution of the query to finish this exercise.

Modifying Local Variables

You can also modify values of variables from the Locals window.

To modify a local variable:

1 In the debugger Source window, set a breakpoint at the following line in the **selecter( )** method of the **serverMain** class:

```
String sql = "select name, home from xmp where
   id=?";
```

2 Step past this line in the execution.

3 Open the Locals window. Select the *id* variable, and select Local→Modify. Alternatively, you can set the cursor on the line and press **Enter**.

4 Enter a value of 2 in the text box, and click **OK** to confirm the new value. The *id* variable is set to 2 in the Locals window.

5 From the Source window, press F5 to complete execution of the query. In the Interactive SQL Data window, an error message displays indicating that no rows were found.

Inspecting Static Variables

You can also inspect the values of class-level variables (static variables).

To inspect a static variable:

1 From the debugger Source window, select Window→Classes. The Classes window is displayed.

2 Select a class in the left box. The methods and static variables of the class are displayed in the boxes on the right.

3 Select Static→Inspect. The Inspect window is displayed. It lists the variables available for inspection.

CHAPTER 7

Reference Topics

This chapter presents information on several reference topics.

These topics are discussed:

Assignments

This section defines the rules for assignment between SQL data items whose datatypes are Java-SQL classes.

Each assignment transfers a *source instance* to a *target data item*:

- For an **insert** statement specifying a table that has a Java-SQL column, refer to the Java-SQL column as the target data item and the insert value as the source instance.
- For an **update** statement that updates a Java-SQL column, refer to the Java-SQL column as the target data item and the update value as the source instance.
- For a **select** or **fetch** statement that assigns to a variable or parameter, refer to the variable or parameter as the target data item and the retrieved value as the source instance.

Note If the source is a variable or parameter, then it is a reference to an object in the Java VM. If the source is a column reference, which contains a serialization, then the rules for column references (see **Java-SQL Column References on page 172**) yield a reference to an object in the Java VM. Thus, the source is a reference to an object in the Java VM.

Assignment Rules at Compile-Time

1 Define **SC** and **TC** as compile-time class names of the source and target. Define **SC_T** and **TC_T** as classes named **SC** and **DT** in the database associated with the target. Similarly, define **SC_S** and **TC_S** as classes named **SC** and **DT** in the database associated with the source.

2 **SC_T** must be the same as **TC_T** or a subclass of **TC_T**.

Assignment Rules at Runtime

Assume that **DT_SC** is the same as **DT_TC** or its subclass.

- Define **RSC** as the runtime class name of the source value. Define **RSC_S** as the class named **RSC** in the database associated with the source. Define **RSC_T** as the name of a class **RSC_T** installed in the database associated with the target. If there is no class **RSC_T**, then an exception is raised. If **RSC_T** is neither the same as **TC_T** nor a subclass of **TC_T**, then an exception is raised
- If the databases associated with the source and target are not the same database, then the source object is serialized by its current class, **RSC_S**, and that serialization is deserialized by the class **RSC_T** that it will be associated with in the database associated with the target.
- If the target is a SQL variable or parameter, then the source is copied to the target.
- If the target is a Java-SQL column, then the source is serialized, and that serialization is copied to the client.

Allowed Conversions

You can use **convert** to change the expression datatype in these ways:

- Convert Java types where the Java datatype is a Java object type to the SQL datatype shown in “Datatype Mapping Between Java and SQL” on page 166. The action of the **convert** function is the mapping implied by the Java-SQL mapping
- Convert SQL datatypes to Java types shown in “Datatype Mapping Between Java and SQL” on page 166. The action of the **convert** function is the mapping implied by the SQL-Java mapping.
- Convert any Java-SQL class installed in the SQL system to any other Java-SQL class installed in the SQL system if the compile-time datatype of the expression (source class) is a subclass or superclass of the target class. Otherwise, an exception is raised.

The result of the conversion is associated with the current database.

See “Using the SQL convert function for Java subtypes,” for a discussion of the use of the **convert** function for Java subtypes.

Transferring Java-SQL Objects to Clients

When a value whose datatype is a Java-SQL object type is transferred from Adaptive Server to a client, the data conversion of the object depends on the client type:

- If the client is an **isql** client, the **toString( )** method of the object is invoked and the result is truncated to varchar(50), which is transferred to the client.
- If the client is a Java client that uses jConnect 4.0 or later, the server transmits the object serialization to the client. This serialization is seamlessly deserialized by jConnect to yield a copy of the object.
- If the client is a **bcp** client:
  - If the object is a column declared as **in row**, the serialized value contained in the column is transferred to the client as a varbinary(255) value.
  - Otherwise, the serialized value of the object (the result of the **writeObject** method of the object) is transferred to the client as an image value.

Supported Java API Packages, Classes, and Methods

Adaptive Server supports many but not all classes and methods in the Java API. In addition, Adaptive Server may impose security restrictions and implementation limitations. For example, Adaptive Server does not support all of the thread creation and manipulation facilities of **java.lang.Thread**.

The supported packages are installed with Adaptive Server and are always available. They cannot be installed by the user.

This section lists:

- Supported Java packages and classes
- Unsupported Java packages
- Unsupported **java.sql** methods

Supported Java Packages and Classes

- **java.io**
  - **Externalizable**
  - **DataInput**
  - **DataOutput**
  - **ObjectInputStream**
  - **ObjectOutputStream**
  - **Serializable**
- **java.lang**
- **java.lang.reflect**
- **java.math**
- **java.sql** – the JDBC driver, see “Unsupported java.sql Methods” on page 159
- **java.text**
- **java.util**
- **java.util.zip**

Unsupported Java Packages

- **java.applet**
- **java.awt**
- **java.awt.datatransfer**
- **java.awt.event**
- **java.awt.image**
- **java.awt.peer**
- **java.beans**
- **java.rmi**
- **java.rmi.dgc**
- **java.rmi.registry**
- **java.rmi.server**
- **java.security**
- **java.security.acl**
- **java.security.interfaces**
- **java.net**

Unsupported *java.sql* Methods

- **Connection.commit( )**
- **Connection.getMetaData( )**
- **Connection.nativeSQL( )**
- **Connection.rollback( )**
- **Connection.setAutoCommit( )**
- **Connection.setCatalog( )**
- **Connection.setReadOnly( )**
- **Connection.setTransactionIsolation( )**
- **DatabaseMetaData.*** (all methods)
- **PreparedStatement.setAsciiStream( )**

- **PreparedStatement.setUnicodeStream( )**
- **PreparedStatement.setBinaryStream( )**
- **ResultSetMetaData.getCatalogName( )**
- **ResultSetMetaData.getSchemaName( )**
- **ResultSetMetaData.getTableName( )**
- **ResultSetMetaData.isCaseSensitive( )**
- **ResultSetMetaData.isReadOnly( )**
- **ResultSetMetaData.isSearchable( )**
- **ResultSetMetaData.isWritable( )**
- **Statement.getMaxFieldSize( )**
- **Statement.setMaxFieldSize( )**
- **Statement.setCursorName( )**
- **Statement.setEscapeProcessing( )**
- **Statement.getQueryTimeout( )**
- **Statement.setQueryTimeout( )**

Invoking SQL from Java

Adaptive Server supplies a native JDBC driver, **java.sql**, that implements JDBC 1.1 specifications. It is described at *http://www.javasoft.com*. **java.sql** enables Java methods executing in Adaptive Server to perform SQL operations.

Special Considerations

java.sql.DriverManager.getConnection( ) accepts these URLs:

- null
- "" (the null string)
- **jdbc:default:connection**

When invoking SQL from Java some restrictions apply:

- A SQL query that is performing update actions (**update**, **insert**, or **delete**) cannot use the facilities of **java.sql** to invoke other SQL operations that also perform update actions.
- Triggers that are fired by SQL using the facilities of **java.sql** cannot generate result sets.
- **java.sql** cannot be used to execute extended stored procedures or remote stored procedures.

Transact-SQL Commands from Java Methods

You can use certain Transact-SQL commands in Java methods called within the SQL system. Table 7-1 lists Transact-SQL commands and whether or not you can use them in Java methods.

Table 7-1: Support status of Transact-SQL commands

Command	Status
alter database	Not supported.
alter role	Not supported.
alter table	Supported.
begin ... end	Supported.
begin transaction	Not supported.
break	Supported.

Command	Status
case	Supported.
checkpoint	Not supported.
commit	Not supported.
compute	Not supported.
connect - disconnect	Not supported.
continue	Supported.
create database	Not supported.
create default	Not supported.
create existing table	Not supported.
create index	Not supported.
create procedure	Not supported.
create role	Not supported.
create rule	Not supported.
create schema	Not supported.
create table	Supported.
create trigger	Not supported.
create view	Not supported.
cursors	Not supported. Only “server cursors” are supported, that is, cursors that are declared and used within a stored procedure.
dbcc	Not supported.
declare	Supported.
disk init	Not supported.
disk mirror	Not supported.
disk refit	Not supported.
disk reinit	Not supported.
disk remirror	Not supported.
disk unmirror	Not supported.
drop database	Not supported.
drop default	Not supported.
drop index	Not supported.
drop procedure	Not supported.
drop role	Not supported.
drop rule	Not supported.
drop table	Supported.

Command	Status
drop trigger	Not supported.
drop view	Not supported.
dump database	Not supported.
dump transaction	Not supported.
execute	Supported.
goto	Supported.
grant	Not supported.
group by and **having** clauses	Supported.
if…else	Supported.
insert table	Supported.
kill	Not supported.
load database	Not supported.
load transaction	Not supported.
online database	Not supported.
order by Clause	Supported.
prepare transaction	Not supported.
print	Not supported.
raiserror	Supported.
readtext	Not supported.
return	Supported.
revoke	Not supported.
rollback trigger	Not supported.
rollback	Not supported.
save transaction	Not supported.
set	See Table 7-2 for **set** options.
setuser	Not supported.
shutdown	Not supported.
truncate table	Supported.
union Operator	Supported.
update statistics	Not supported.
update	Supported.
use	Not supported.
waitfor	Supported.
where Clause	Supported.
while	Supported.

Command	Status
writetext	Not supported.

Table 7-2 lists **set** command options and whether or not you can use them in Java methods.

Table 7-2: Support status of set command options

set Command Option	Status
ansinull	Supported.
ansi_permissions	Supported.
arithabort	Supported.
arithignore	Supported.
chained	Not supported. See Note 1.
char_convert	Not supported.
cis_rpc_handling	Not supported
close on endtran	Not supported
cursor rows	Not supported
datefirst	Supported
dateformat	Supported
fipsflagger	Not supported
flushmessage	Not supported
forceplan	Supported
identity_insert	Supported
language	Not supported
lock	Supported
nocount	Supported
noexec	Not supported
offsets	Not supported
or_strategy	Supported
parallel_degree	Supported. See Note 2.
parseonly	Not supported
prefetch	Supported
process_limit_action	Supported. See Note 2.
procid	Not supported
proxy	Not supported
quoted_identifier	Supported
replication	Not supported
role	Not supported
rowcount	Supported

set Command Option	Status
scan_parallel_degree	Supported. See Note2.
self_recursion	Supported
session_authorization	Not supported
showplan	Supported
sort_resources	Not supported
statistics io	Not supported
statistics subquerycache	Not supported
statistics time	Not supported
string_rtruncation	Supported
table count	Supported
textsize	Not supported
transaction iso level	Not supported. See Note 1.
transactional_rpc	Not supported

Note (1) **set** commands with options chained or transaction isolation level are allowed only if the setting that they specify is already in effect. That is, this kind of **set** command is allowed if it has no effect. This is done to support common coding practises in stored procedures.

Note (2) **set** commands pertaining to parallel degree are allowed but have no effect. This supports the use of stored procedures that set the parallel degree for other contexts.

Datatype Mapping Between Java and SQL

Adaptive Server maps SQL datatypes to Java types (SQL-Java datatype mapping) and Java scalar types to SQL datatypes (Java-SQL datatype mapping). Table 7-3 shows SQL-Java datatype mapping.

Table 7-3: Mapping SQL datatypes to Java types

SQL type	Java type
char	String
varchar	String
nchar	String
nvarchar	String
text	String
numeric	java.math.BigDecimal
decimal	java.math.BigDecimal
money	java.math.BigDecimal
smallmoney	Java.math.BigDecimal
bit	boolean
tinyint	byte
smallint	short
integer	int
real	float
float	double
double precision	double
binary	byte[ ]
varbinary	byte[ ]
image	byte[ ]
datetime	java.sql.Timestamp
smalldatetime	java.sql.Timestamp

Table 7-4 shows Java-SQL datatype mapping.

Table 7-4: Mapping Java scalar types to SQL datatypes

Java Scalar type	SQL type
boolean	bit
byte	tinyint
short	smallint
int	integer
long	integer
float	real

Java Scalar type	SQL type
double	double

Java-SQL Identifiers

Description

Java-SQL identifiers are Java identifiers that can be referenced in SQL. They are a subset of Java identifiers.

Syntax

java_sql_identifier ::= alphabetic character | underscore (_) symbol
[alphabetic character | arabic numeral | underscore(_) symbol |
dollar ($) symbol]

Usage

- Java-SQL identifiers can be a maximum of 255 bytes in length if they are surrounded by quotation marks. Otherwise, they must be 30 bytes or less.
- The first character of the identifier must be either an alphabetic character (uppercase or lowercase) or the underscore (_) symbol. Subsequent characters can include alphabetic characters (uppercase or lowercase), numbers, the dollar ($) symbol, or the underscore (_) symbol.
- Java-SQL identifiers are always case sensitive.

Delimited Identifiers

- Delimited identifiers are object names enclosed in double quotes. Using delimited identifiers for Java-SQL identifiers allows you to avoid certain restrictions on the names of Java-SQL identifiers.

  Note You can use double quotes with Java-SQL identifiers whether the **set quoted_identifier** option is **on** or **off**.

- Delimited identifiers allow you to use SQL reserved words for packages, classes, methods, and so on. Each time you use the delimited identifier in a statement, you must enclose it in double quotes. For example:

```
create table t1
   (c1 char(12)
   c2 p1.”select”.p2.”jar”)
```

- Double quotes surround only individual Java-SQL identifiers, not the fully qualified name.

See also

For additional information about identifiers, see Chapter 5, “Transact-SQL Topics,” in the *Reference Manual*.

Java-SQL Class and Package Names

Description

To reference a Java-SQL class or package, use the following syntax:

Syntax

java_sql_class_name ::= [*java_sql_package_name.*]*java_sql_identifier*

java_sql_package_name ::=
[*java_sql_package_name.*]*java_sql_identifier*

Parameters

java_sql_class_name
The fully qualified name of a Java-SQL class in the current database.

java_sql_package_name
The fully qualified name of a Java-SQL package in the current database.

Usage

For Java-SQL class names:

- A class name reference always refers to a class in the current database.
- If you specify a Java-SQL class name without referencing the package name, only one Java-SQL class of that name must exist in the current database, and its package must be the default (anonymous) package.
- If a SQL user-defined datatype and a Java-SQL class possess the same sequence of identifiers, Adaptive Server uses the SQL user-defined datatype name and ignores the Java-SQL class name

For Java-SQL package names:

- If you specify a Java-SQL subpackage name, you must reference the subpackage name with its package name:

  java_sql_package_name.java_sql_subpackage_name

- Use Java-SQL package names only as qualifiers for class names or subpackage names and to delete packages from the database using the **remove java** command.

Java-SQL Column Declarations

Description

To declare a Java-SQL column when you create or alter a table, use the following syntax:

Syntax

java_sql_column ::= column_name java_sql_class_name

Parameters

java_sql_column
Specifies the syntax of Java-SQL column declarations.

column_name
The name of the Java-SQL column.

java_sql_class_name
The name of a Java-SQL class in the current database. This is the “declared class” of the column.

Usage

- The declared class must implement either the **Serializable** or **Externalizable** interface.
- A Java-SQL column is always associated with the current database.
- A Java-SQL column cannot be specified as:
  - **not null**
  - **unique**
  - A primary key

See also

You use a Java-SQL column declaration only when you create or alter a table. See the **create table** and **alter table** information in the *Reference Manual*.

Java-SQL Variable Declarations

Description

Use Java-SQL variable declarations to declare variables and stored procedure parameters for datatypes that are Java-SQL classes.

Syntax

java_sql_variable ::= *@variable_name java_sql_class_name*

java_sql_parameter ::= *@parameter_name java_sql_class_name*

Parameters

java_sql_variable
Specifies the syntax of a Java-SQL variable in a SQL stored procedure.

java_sql_parameter
Specifies the syntax of a Java-SQL parameter in a SQL stored procedure.

java_sql_class_name
The name of a Java-SQL class in the current database.

Usage

A *java_sql_variable* or *java_sql_parameter* is always associated with the database containing the stored procedure.

See also

Refer to the *Reference Manual* for more information about variable declarations.

Java-SQL Column References

Description

To reference a field or method of a class or class instance, use the following syntax:

Syntax

column_reference ::=
[[[*database_name*.]*owner*.]*table_name*.]*column_name*
| *database_name*..*table_name*.*column_name*

Parameters

column_reference

A reference to a column whose datatype is a Java-SQL class.

Usage

- If the value of the column is null, then the column reference is also null.
- If the value of the column is a Java serialization, S, and the name of its class is **CS**, then:
  - If the class **CS** does not exist in the current database or if **CS** is not the name of a class in the database associated with the serialization, then an exception is raised.

    Note The database associated with the serialization is normally the database that contains the column. Serializations contained in work tables and in temporary tables created with "insert into #tempdb" are, however, associated with the database in which the serialization was stored originally.

  - The value of the column reference is:

    ```
    CSC.readObject(S)
    ```

    where CSC is the column reference. If the expression raises an uncaught Java exception, then an exception is raised.

    The expression yields a reference to an object in the Java VM, which is associated with the database associated with the serialization.

Java-SQL Member References

Description

References a field or method of a class or class instance.

Syntax

member_reference ::= *class_member_reference* |
instance_member_reference

class_member_reference ::= *java_sql_class_name.method_name*

instance_member_reference ::= *instance_expression>>member_name*

instance_expression ::= *column_reference* | *variable_name*
| *parameter_name* | *method_call* | *member_reference*

member_name ::= *field_name* | *method_name*

Parameters

member_reference
An expression that describes a field or method of a class or object.

class_member_reference
An expression that describes a static method of a Java-SQL class.

instance_member_reference
An expression that describes a static or dynamic method or field of a Java-SQL class instance.

java_sql_class_name
A fully qualified name of a Java-SQL class in the current database.

instance_expression
An expression whose datatype is a Java-SQL class.

member_name
The name of a field or method of the class or class instance.

Usage

- If a member references a field of a class instance, the instance has a null value, and the Java-SQL member reference is the target of a **fetch**, **select**, or **update** statement, then an exception is raised.

  Otherwise, the Java-SQL member reference has the null value.

- The double angle (>>) and dot (.) qualification takes precedence over any operator, such as the addition (+) or equal to (=) operator, for example:

  ```
  X>>A1>>B1 + X>>A1>>B2
  ```

  In this expression, the addition operation is performed after the members have been referenced.

- The field or method designated by a member reference is associated with the same database as that of its Java-SQL class or instance of its Java-SQL class.

If the Java type of a member reference is one of the Java scalar types (such as boolean, byte, and so on), then the corresponding SQL datatype of the reference is obtained by mapping the Java type to its equivalent SQL type.

If the Java type of a member reference is an object type, then the SQL datatype is the same Java object type or class.

Java-SQL Method Calls

Description

To invoke a Java-SQL method, which returns a single value, use the following syntax:

Syntax

method_call ::= *member_reference* ([*parameters*])
| new *java_sql_class_name* ([*parameters*])

parameters ::= *parameter* [(, *parameter*)...]

parameter ::= *expression*

Parameters

method_call
An invocation of a class method, instance method, or class constructor. A method call can be used in an expression where a non-constant value of the method's datatype is required.

member_reference
A member reference that denotes a method.

parameters
The list of parameters to be passed to the method. If there are no parameters, include empty parentheses.

Usage

Method Overloading

- When there are methods with the same name in the same class or instance, the issue is resolved according to Java method overloading rules.

Datatype of Method Calls

- The datatype of a method call is determined as follows:
  - If a method call specifies **new**, its datatype is that of its Java-SQL class.
  - If a method call specifies a member reference that denotes a type-valued method, then the datatype of the method call is that type.
  - If a method call specifies a member reference that denotes a void static method, then the datatype of the method call is SQL integer.
  - If a method call specifies a member reference that denotes a void instance method of a class, then the datatype of the method call is that of the class.
- If you want to include a parameter in a member reference when the parameter is a Java-SQL instance associated with another database, you must ensure that the class name associated with the Java-SQL instance is included in both databases. Otherwise, an exception is raised.

Runtime Results

- The runtime result of a method call is as follows:
  - If a method call specifies a member reference whose runtime value is null (that is, a reference to a member of a null instance), then the result is null.
  - If a method call specifies a member reference that denotes a type-valued method, then the result is the value returned by the method.
  - If a method call specifies a member reference that denotes a void static method, then the result is the null value.
  - If a method call specifies a member reference that denotes a void instance method of an instance of a class, then the result is a reference to that instance.
  - The method call and result of the method call are associated with the same database.
  - Adaptive Server does not pass the null value as the value of a parameter to a method whose Java type is scalar.

Glossary

This glossary describes Java and Java-SQL terms used in this book. For a description of Adaptive Server and SQL terms, refer to the *Adaptive Server Glossary*.

assignment A generic term for the data transfers specified by **select**, **fetch**, **insert**, and **update** T-SQL commands. An assignment sets a source value into a target data item.

associated JAR If a class/jar is installed with **installjava** and the **-jar** option, then the JAR is retained in the database and the class is linked in the database with the associated jar. See **retained JAR**.

bytecode The compiled form of Java source code that is run by the Java VM.

class A class is the basic element of Java programs, containing a set of variable declarations and methods. A class is the master copy that determines the behavior and attributes of each instance of that class. See **class instance**.

class file A file of type "class" (for example, *myclass.class*) that contains the compiled bytecode for a Java class. See **Java file** and **Java archive (JAR)**.

class instance An single copy of each of the fields of the class. Class instances are strongly typed by the class name.

datatype mapping Conversions between Java and SQL datatypes.

declared class The declared datatype of a Java-SQL data item. It is either the datatype of the runtime value or a supertype of it.

document type declaration (DTD) In XML, every valid document has a DTD that describes the elements available in that document type. A DTD can be embedded in the XML document or referenced by it.

Extensible Markup Language (XML) A metalanguage designed for Web applications that lets you define your own markup tags and attributes for different kinds of documents. XML is a subset of SGML.

Extensible Style Language (XSL) A markup language designed to format XML documents into HTML or other XML documents with different attributes and tags.

externalization

An externalization of a Java instance is a byte stream that contains sufficient information for the class to reconstruct the instance. Externalization is defined by the externalizable interface. All Java-SQL classes must be either externalizable or serializable. See **serialization**.

friendly

A **friendly** method can be called only by methods of other classes in the same package.

Hypertext Markup Language (HTML)

A subset of SGML designed for the Web.

installed classes

Installed Java classes and methods have been placed in the Adaptive Server system by the **installjava** utility.

instance

A particular copy of a class. An object that is contained in the Java VM. See **class instance**.

interface

A unique type of class that lets a class inherit particular methods.

Java archive (JAR)

A platform-independent format for collecting classes in a single file.

Java Database Connectivity (JDBC)

A Java-SQL API that is a standard part of the Java Class Libraries that control Java application development. JDBC provides capabilities similar to those of ODBC.

Java Development Kit (JDK)

A toolset from Sun Microsystems that allows you to write and test Java programs from the operating system.

Java file

A file of type "java" (for example, *myfile.java*) that contains Java source code. See **class file** and **Java archive (JAR)**.

Java object

An instance of a Java class that is contained in the storage of the Java VM. Java instances that are referenced in SQL are either values of Java columns or Java objects.

Java-SQL column

A SQL column whose datatype is a Java-SQL class.

Java-SQL class

A public Java class that has been installed in the Adaptive Server system. It consists of a set of variable definitions and methods.

A class instance consists of an instance of each of the fields of the class. Class instances are strongly typed by the class name.

A subclass is a class that is declared to extend (at most) to one other class. That other class is called the direct superclass of the subclass. A subclass has all of the variables and methods of its direct and indirect superclasses, and may be used interchangeably with them.

Java-SQL datatype mapping

Conversions between Java and SQL datatypes. See “Datatype Mapping Between Java and SQL” on page 166.

Java-SQL variable

A SQL variable whose datatype is a Java-SQL class.

Java Virtual Machine (Java VM)

The Java interpreter that processes Java in the server. It is invoked by the SQL implementation.

mappable

A Java datatype is mappable if it is either:

- Listed in the first column of Table 7-3 on page 166, or
- A public Java-SQL class that is installed in the Adaptive Server system.

A SQL datatype is mappable if it is either:

- Listed in the first column of Table 7-4 on page 166, or
- A public Java-SQL class that is built-in or installed in the Adaptive Server system.

A Java method is mappable if all of its parameter and result datatypes are mappable.

method

A set of instructions, contained in a Java class, for performing a task. A method can be declared static, in which case it is called a class method. Otherwise, it is an instance method. Class methods can be referenced by qualifying the method name with either the class name or the name of an instance of the class. Instance methods are referenced by qualifying the method name with the name of an instance of the class. The method body of an instance method can reference the variables local to that instance

narrowing conversion

A Java operation for converting a reference to a class instance to a reference to an instance of a subclass of that class. This operation is written in SQL with the **convert** function. See also **widening conversion**.

package

A package is a set of related classes. A class either specifies a package or is part of an anonymous default package. A class can use Java **import** statements to specify other packages whose classes can then be referenced.

procedure

An SQL stored procedure, or a Java method with a *void* result type.

public

Public fields and methods, as defined in Java.

retained JAR

A JAR that is installed by the **installjava** utility with the **-jar** option. A retained JAR is associated in the database with the classes it contains.

serialization — A serialization of a Java instance is a byte stream containing sufficient information to identify its class and reconstruct the instance. All Java-SQL classes must be either externalizable or serializable. See **externalization**.

SQL92 — The current SQL standard.

SQL3 — The working draft for the next revision of the SQL standard.

SQL-Java datatype mapping — See **datatype mapping**.

subclass — A class below another class in a hierarchy. It inherits attributes and behavior from classes above it. A subclass may be used interchangeably with its superclasses. The class above the subclass is its direct superclass. See **superclass**, **narrowing conversion**, and **widening conversion**.

superclass — A class above one or more classes in a hierarchy. It passes attributes and behavior to the classes below it. It may not be used interchangeably with its subclasses. See **subclass**, **narrowing conversion**, and **widening conversion**.

synonymous classes — Java-SQL classes that have the same fully qualified name but are installed in different databases.

Unicode — A 16-bit character set defined by ISO 10646 that supports many languages.

valid document — In XML, a valid document has a DTD and adheres to it. It is also a **well-formed document**.

variable — In Java, a variable is local to a class, to instances of the class, or to a method. A variable that is declared static is local to the class. Other variables declared in the class are local to instances of the class. Those variables are called fields of the class. A variable declared in a method is local to the method.

visible — A Java class that has been installed in a SQL system is visible in SQL if it is declared **public**; a field or method of a Java instance is visible in SQL if it is both **public** and mappable. Visible classes, fields, and methods can be referenced in SQL. Other classes, fields, and methods cannot, including classes that are **private**, **protected**, or **friendly**, and fields and methods that are either **private**, **protected**, or **friendly**, or are not mappable.

well-formed document — In XML, the necessary characteristics of a well-formed document include: all elements with both start and end tags, attribute values in quotes, all elements properly nested.

widening conversion — A Java operation for converting a reference to a class instance to a reference to an instance of a superclass of that class. This operation is written in SQL with the **convert** function. See also **narrowing conversion**.

Symbols

A

B

C

D

E

G

H

I

J

M

T

U

V

W

X

Z