SYBASE®
An SAP Company

Java in Adaptive Server Enterprise

Adaptive Server® Enterprise

15.7

DOCUMENT ID: DC31652-01-1570-01

LAST REVISED: September 2011

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

CHAPTER 1

An Introduction to Java in the Database

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

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 allows you to:

- Invoke Java methods in the database
- Store Java classes as datatypes
- Store and query XML in the database

Invoking Java methods in the database

You can install Java classes in Adaptive Server, and then invoke the static methods of those classes in two ways:

- You can invoke the Java methods directly in SQL.
- You can wrap the methods in SQL names and invoke them as you would standard Transact-SQL stored procedures.

Invoking Java methods directly in SQL

The methods of an object-oriented language correspond to the functions of a procedural language. You can invoke methods stored in the database by referencing them, with name qualification, on instances for instance methods, and on either instances or classes for static (class) methods. You can invoke the method directly in, for example, Transact-SQL select lists and where clauses.

You can use static methods that return a value to the caller as user-defined functions (UDFs).

Certain restrictions apply when using Java methods in this way:

- If the 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.

Invoking Java methods as SQLJ stored procedures and functions

You can enclose Java static methods in SQL wrappers and use them exactly as you would Transact-SQL stored procedures or built-in functions. This functionality:

- Allows Java methods to return output parameters and result sets to the calling environment.
- Allows you to take advantage of traditional SQL syntax, metadata, and permission capabilities.
- Allows you to invoke SQLJ functions across databases.
- Allows you to use existing Java methods as SQLJ procedures and functions on the server, on the client, and on any SQLJ-compliant, third-party database.
- Complies with Part 1 of the standard specification. See “Standards” on page 7.

Storing 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 a SQL database. 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 standalone 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 datatypes or store the Java objects as untyped binary strings.

Storing and querying XML in the database

Similar to Hypertext Markup Language (HTML), the eXtensible Markup Language (XML) allows you to define your own application-specific markup tags and is thus particularly suited for data interchange.

XML Services in Adaptive Server Enterprise describes the Sybase® native XML processor and the Sybase Java-based XML support, introduces XML in the database, and documents the query and mapping functions that comprise XML Services.

Java components

Adaptive Server lets you plug in commercial, off-the-shelf Java runtime environment (JRE) and Java virtual machine (JVM) components. After configuring Adaptive Server for Java, you can include any standard JVM that supports Java 6 or later. This infrastructure lets you run Java applications configured with the Java solution in Adaptive Server versions prior to 15.0.3 as well as applications created using the Adaptive Server version 15.0.3 and later.

The Java interface for Adaptive Server include the commercial JVM and the Sybase components that support it:

- The pluggable component adaptor/ JVM (PCA/JVM)
- The pluggable component interface (PCI) and the PCI Bridge, which are internal to Adaptive Server

See Chapter 2, “Managing the Java Environment.”

Functional changes in Adaptive Server 15.0.3 and later

With Adaptive Server version 15.0.3, Sybase introduces support for commercial JVMs such as the Sun Java 2 Platform, Standard Edition (J2SE). Adaptive Server version 15.0.2 and earlier provided an internal JVM.

The Adaptive Server PCA/JVM ensures that Java applications created before version 15.0.3 run seamlessly with Java applications you create with Adaptive Server version 15.0.3 and later.

In addition to the changes described in this section, see:

- "Changes for static variables for Adaptive Server 15.0.3 and later" on page 63
- "Changes for static variables for the Cluster Edition" on page 64

Changes in class distribution

The Java runtime classes delivered with Adaptive Server 15.0.2 and earlier was a limited subset of the Java 1.2 release. Adaptive Server no longer provides the runtime classes. Rather, the JVM uses the runtime classes delivered as part of the commercial JRE.

In general, Java classes from later versions can be presumed to be backwards compatible with earlier versions. However, certain methods or classes marked "deprecated" in earlier versions may no longer be compatible with later versions. Make sure that any deprecated classes or methods used by your applications are still supported and unchanged in later versions of Java.

Adaptive Server version 15.0.2 and earlier included a *runtime.zip* file in the *$SYBASE/$SYBASE_ASE/lib* directory. This file included the Adaptive Server specific classes, JDBC classes required for driver support, and a subset of the standard Java classes.

Adaptive Server 15.0.3 replaces the *runtime.zip* file with the *sybasert.jar* (which contains the Sybase Java classes required by the PCA/JVM) and uses the *rt.jar* to provide the standard Java class set. *sybasert.jar* is located in *$SYBASE/ASE-15_0/lib/pca*, and *rt.jar* is located in the Java distribution in *$SYBASE/shared/<jre_directory>/lib*, where *jre_directory* is a name specific to your platform.

The PCA/JVM runs in headless mode

Classes and methods requiring user interaction were excluded from the Java distribution provided by Adaptive Server 15.0.2 and earlier. Because the PCA/JVM uses the standard class distribution, these classes are now available. To prevent users from invoking methods that require user interaction, the PCA/JVM always runs in headless mode.

Changes in memory management

Adaptive Server 15.0.2 and earlier used a memory management system consisting of three distinct heaps: a global fixed heap, a shared class heap, and a process object heap. Adaptive Server 15.0.3 and later uses a single PCI memory pool. Any existing configuration values from 15.0.2 and earlier are ignored by Adaptive Server 15.0.3. You must specify the total memory for the PCI subsystem using the pci memory size configuration parameter. See Chapter 2, “Managing the Java Environment.”

If you are transitioning from Adaptive Server version 15.0.2 and earlier, you may need to change the default size of the PCI memory pool. The life cycle of classes and garbage collection algorithms used by commercial JVMs differs significantly from that of the Sybase internal JVM. Once the size of the PCI memory pool is appropriately configured, you should see no difference in behavior.

Changes in ClassLoader behavior

In Adaptive Server version 15.0.3 and later, ClassLoader behavior conforms to JVM specifications for the verification of classes during loading.

In Adaptive Server version 15.0.2 and earlier, references to additional classes within the class being loaded were checked but not fully resolved. For example, if class A referred to class B within a method, the ClassLoader did not check that class B was actually available. Thus, a class could successfully load without satisfying all of its dependencies. An exception would be raised only when the method that requiring the unsatisfied dependency was encountered.

The ClassLoader for all commercial JVM implementations performs the full class verification when the initial class is loaded. As a result, a class with unsatisfied dependencies does not load, an Unhandled Java Exception is raised, and the Java stack trace lists the error as “java.lang.NoClassDefFoundError.”

This means that, in rare instances, a class that loads successfully in Adaptive Server 15.0.2 and earlier may not load in Adaptive Server 15.0.3 and later unless a full set of user and Java-supplied classes is provided so that all dependencies can be satisfied.

Standards

The ANSI SQL standards specify SQL extensions for using Java facilities in SQL. The Java-SQL specifications are in the SQL standard, "Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)." This standard is referred to informally as "SQLJ."

Sybase supports the SQLJ specifications for Java routines, and provides equivalent facilities for Java types. In addition, Sybase extends the standard. For example, Adaptive Server allows you to reference Java methods and classes directly in SQL.

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 using any commercial JVM that supports Java 6 or later.
- Call Java functions (methods) directly from SQL statements.
- Wrap Java methods in SQL aliases and call them as standard SQL stored procedures and built-in functions.
- Access SQL data from Java using an internal JDBC driver.
- Use Java classes as SQL datatypes.
- Save instances of Java classes in tables.

- Generate XML-formatted documents from raw data stored in Adaptive Server databases and, conversely, store XML documents and data extracted from them in Adaptive Server databases.
- Debug Java classes running in the database.

How are Java instructions stored in the database?

Java is an object-oriented language. Its instructions 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 provides 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?

When Adaptive Server encounters a Java statement within an executing SQL statement, the server invokes the JVM to execute the statement. If the JVM is already running, the Java invocation is forwarded to it; if this is the first Java request, the JVM starts automatically. The JVM locates and loads the class identified by the Java statement and executes the byte code.

Which Java Virtual Machines (JVMs) are supported?

The Adaptive Server Java framework has been designed to work with any standard JVM that supports Java 6 or later. Adaptive Server version 15.0.3 has been certified with Java 6 version that is included in the *$SYBASE/shared* directory. Classes compiled by earlier versions of Java will continue to run correctly under later versions of Java.

What is headless mode?

Java in Adaptive Server runs in headless mode, which means that display devices, keyboards, and mice are not used. Although all classes in the standard Java distribution are available to the user, certain methods that expect user input or output devices are not supported.

What about JDBC?

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

Adaptive Server provides a native 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 driver permits Java classes installed in a database to use JDBC classes that execute SQL statements.

How can Java and SQL be used 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 you can include Java operations in a SQL statement.
- *Java methods as* SQLJ *stored procedures and functions* – you create a SQLJ alias for Java static methods, so that you can invoke them as standard SQL stored procedures and user-defined functions (UDFs).
- *Java classes become user-defined datatypes* – you store Java class instances 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.

What is the Java API?

The Java Application Programming Interface (API) is a basic set of functionality defined by Sun Microsystems. It 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, and is the foundation for all user-defined classes created for individual user applications.

Which Java classes are supported in the Java API?

Adaptive Server supports all standard Java classes in the database. Because Java in the database runs in headless mode (see “What is headless mode?” on page 9), certain methods expecting user input or output devices raise a Java exception.

Can user-defined classes be installed in the database?

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 data and methods to operate on data. 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).

Can data be accessed 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.

Adaptive Server provides an internal JDBC driver, which permits Java classes installed in a database to use JDBC classes that execute SQL statements.

Can the same classes be used on the client and the server?

You can create Java classes that can be used on different levels of an enterprise application. You can integrate the same Java class into either the client application, a middle tier, or the database.

Take care that classes used in different tiers, or in the same tier over time, remain compatible or are knowingly made incompatible so that behavior is consistent across the application. See the Java documentation on the serialVersionUID in the java.io.Serializable class for details.

How to use Java classes in SQL

Using user-defined Java classes is a three-step activity:

1 Write or acquire a set of Java classes that you want to use as SQL datatypes, or as SQL aliases for static methods.

2 Install those classes in the Adaptive Server database.

Note Classes included in the Java distribution are always available and do not need to be installed in the database prior to use.

3 Use those classes in SQL code:

- Invoke static methods directly 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 fields or methods of Java-SQL columns, variables, or parameters.
- Wrap static methods in SQL aliases and use them as stored procedures or functions.

Where can information about Java in the database be found?

There are many books about Java and Java in the database. The most recent Java language specification is located on the Sun Web site.

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 perform these actions 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.
- Use the Java Native Interface (JNI).
- Use Java objects as parameters sent to a remote procedure call or received from a remote procedure call. They do not translate correctly.

Sybase recommends that you do not use nonfinal static variables in methods referenced by Java-SQL functions, SQLJ functions, or SQLJ stored procedures. The values returned for these variables may be unreliable as the scope of the static variable is implementation-dependent.

CHAPTER 2

Managing the Java Environment

Topic	Page
Components of the Java environment	13
When to change configuration values	19
Changing configuration values in a running server	21
Restoring default configuration values to sybpcidb	23
Using monitor tables to display information about the PCI Bridge	24

You can plug in off-the-shelf, standard Java JRE and JVM components such as J2SE, to Adaptive Server. This chapter describes the Sybase components that support Java and how to change default configuration values.

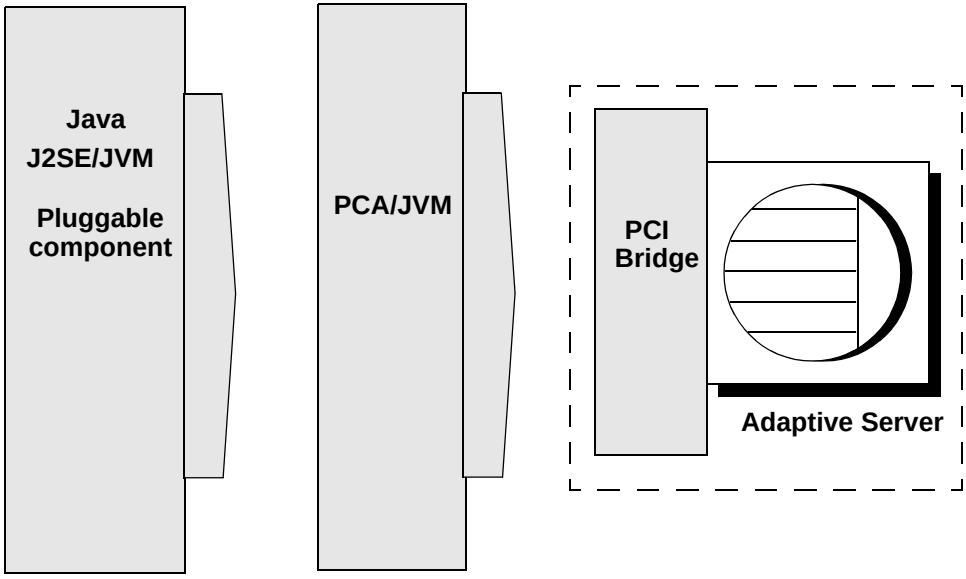
The Adaptive Server Java framework has been designed to work with any standard JVM that supports Java 6 or later. ASE 15.5 has been certified with the Java 6 version that is included in the *$SYBASE/shared* directory. Classes compiled by earlier versions of Java continue to run correctly under later versions of Java.

The JVM is independent of Adaptive Server. You can change or upgrade your Java applications to take advantage of new Java functionality as it becomes available.

Components of the Java environment

Figure 2-1 shows the components that make up the Adaptive Server Java environment.

Figure 2-1: Java components



The JVM pluggable component

The JVM plug-in is a dynamically loaded module that is engineered, supported, and installed on your platform independently from Adaptive Server. To Adaptive Server, the plug-in is a “black-box” application and not Sybase-supported technology: the JVM plug-in issues Java result sets, which are translated by the PCI Bridge, which then sends the translated result sets to Adaptive Server.

Because the JVM plug-in is controlled by the PCA/JVM, it is indirectly connected to Adaptive Server. You can install, upgrade, and start the JVM plug-in independently of Adaptive Server.

Typically, Java distributions include one or more JVM implementations. This allows users to select the VM that best corresponds to the performance requirements of individual applications.

- Client applications – on platforms typically used for client applications, the JRE includes a VM that is tuned to reduce start-up time and memory footprint.
- Server applications – on all platforms, the JRE includes a version of the JVM that is designed for maximum program execution speed.

There are many Java distributions, however, these features of Java technology are common to both the client and server VM versions:

- Adaptive compiler – the Java plug-in uses a standard interpreter to launch applications, but analyzes the code as it runs to detect performance bottlenecks, or "hot spots."
- Rapid memory allocation and garbage collection – Java technology provides for rapid memory allocation for objects, and offers a choice of fast, efficient, state-of-the-art garbage collectors.
- Thread synchronization – the Java programming language allows you to use multiple, concurrent paths of program execution (called "threads"). Java technology includes a thread-handling capability that scales readily for use in large, shared-memory multiprocessor servers.

Note Take care when using methods that spawn child threads. java.lang.Thread objects started within a Java method are scheduled at runtime rather than by the Adaptive Server scheduler. If these threads are processor intensive, or spawn large numbers of threads, server performance can degrade due to competition for processor time.

Although the PCA/JVM plug-in can use either the client or server JVM, Sybase recommends that you use the server version to maximize Java method performance by default; the server version is used by the installation process.

See the client-version documentation for information about the appropriate client version for your enterprise.

Pluggable component adapter JVM (PCA/JVM)

The PCA/JVM acts as a broker, managing service requests between the Adaptive Server and the JVM. The PCA/JVM forwards and controls requests in both directions—from the Adaptive Server to the JVM, and from the JVM to the Adaptive Server.

Pluggable component interface (PCI) and the PCI Bridge

The PCI is a generic interface internal to Adaptive Server; it is installed by default when you install or upgrade Adaptive Server. The PCI Bridge, a component internal to the PCI, performs the actual work between Adaptive Server and the JVM plug-in.

The PCI Bridge provides:

- Native thread (process) management
- Memory management
- Synchronization (lock, condition, and event) management
- Data access service support
- Configuration management
- On-demand function dispatching with automatic plug-in loading
- Signal and exception handling
- Platform runtime support
- Dynamic instrumentation facility
- Error message channeling to the Adaptive Server error log

For most scenarios, the default PCI Bridge configuration is appropriate and sufficient. If necessary, and with the advice of Sybase Technical Support, you can use the sp_pciconfig system stored procedure to modify the PCI configuration. sp_pciconfig includes parameters that allow you to list, report, enable, or disable the directives and arguments in sybpcidb. See “Changing configuration values in a running server” on page 21.

The PCI memory pool

The PCI memory pool is allocated all at once when the PCI Bridge initializes; it does not grow after that. It is controlled by Adaptive Server and is governed by the same restrictions as other memory pools—for example, a single allocation cannot exceed 1MB. The default size of the PCI memory pool is 32,768 KB.

Use the enable pci configuration parameter to enable the PCI memory pool when you configure the server for Java. See the installation guide for your platform.

Changing the size of the PCI memory pool

The default size of the PCI memory pool size is adequate for most nonclustered installations. To increase the size of the memory pool, reset the pci memory size configuration parameter.

For example, to set pci memory size to 13800 pages (each page is 2KB), enter:

```
sp_configure "pci memory size", 13800
```

pci memory size is a dynamic configuration parameter; you do not need to restart Adaptive Server for the change to take effect.

If Adaptive Server does not have sufficient memory available to allocate to the memory pool, this configuration change is ignored and the PCI Bridge does not start.

See the *System Administration Guide: Volume 1* for more information about pci memory size.

Java VM memory consumption in multi-engine Adaptive Server

In a multi-engine environment, multiple Adaptive Server tasks can use the Java VM in parallel. As a result, the Java VM requires more memory in a multi-engine environment than in a single-engine environment. As a result, you may need to increase the size of the PCI memory pool based on the types of applications you are running and the number of users executing Java in parallel.

You can allow Adaptive Server to calculate heap sizes, or you can configure them yourself using sp_jreconfig to set the -Xmx and -Xms arguments of the PCA_JVM_JAVA_OPTIONS directive.

To let Adaptive Server configure heap sizes for you, the calculated heap size must be greater than 4MB and you must not set the -Xmx and -Xms arguments. (Adaptive Server uses the values stored in sybpcidb.)

When Adaptive Server configures heap sizes:

- The -Xmx argument of the PCA_JVM_JAVA_OPTIONS directive is set so that the Java heap size is 65% of the PCI memory pool size.
- The -Xms argument is set to the same value as -Xmx.
- 20% of the Java heap size is configured for the young heap generations, also called the Eden space.

The *sybpcidb* database

The sybpcidb database stores configuration information for the PCI Bridge and the PCA/JVM plug-in. You create sybpcidb, install its tables, and create its system stored procedures when you configure the server for Java. See the installation guide for your platform.

The sybpcidb system stored procedures are:

- sp_pciconfig – configures PCI Bridge properties.
- sp_jreconfig – configures PCA/JVM plug-in properties.

sybpcidb tables

The sybpcidb database contains these tables.

User table	Contents
pci_directives	Directive configuration information for the PCI Bridge.
pci_arguments	Argument configuration information for the PCI Bridge.
pci_slotinfo	Information for each slot, including table names for the relevant directives and arguments.
pci_slot_syscalls	The runtime system call configuration information for the runtime dispatching model used by the PCI Bridge.
pca_jre_directives	Directive information specific to the PCA/JVM plug-in.
pca_jre_arguments	Argument information specific to the PCA/JVM plug-in.

See the *Reference Manual: Tables* for more information about sybpcidb. See the *Reference Manual: Procedures* for more information about sp_pciconfig and sp_jreconfig.

How configuration values are organized in *sybpcidb*

Configuration values for the PCI Bridge and the PCA/JVM are stored in sybpcidb and organized in a hierarchy of directives and arguments. Each directive contains one or more arguments; each argument holds a configuration value. Arguments are of these types:

- "switch" arguments – describe properties that can only be enabled or disabled. Switch arguments contain no data. (PCI Bridge and PCA/JVM)
- "number" arguments – contain numeric property values. (PCI Bridge and PCA/JVM)
- "string" arguments – contain string property values. (PCA/JVM only)

- "array" arguments – are a collection of one or more string property values. (PCA/JVM only)

You can enable or disable each directive and each of its arguments. The state of a directive overrides the states of its arguments. For example, suppose a directive has three arguments: "arg1" is enabled, "arg2" is disabled, and "arg3" is disabled.

- If the directive is enabled, each argument retains its base state. That is "arg1" is enabled, "arg2" is disabled, and "arg3" is disabled.
- If the directive is disabled, the disabled state of the directive overrides the base states of the arguments so that "arg1", "arg2", and "arg3" are all disabled.
- However, if the directive is re-enabled, each argument returns to its base state: "arg1" is enabled, "arg2" is disabled, and "arg3" is disabled. This arrangement lets you disable all arguments or return all arguments to their original states with a single command.

When to change configuration values

The default configuration options for the server and for the PCI Bridge and the PCA/JVM are sufficient for most installations. Although you can safely change and manage a few configuration options on your own, most configuration options should not be changed without instructions from Sybase Technical Support.

You can set configuration options:

- At the server level
- For the PCI Bridge
- For the PCA/JVM

Server-level options

Use sp_configure to change and manage these server-level configuration parameters:

- enable pci – enables the PCI Bridge.

- enable java – enables Java in the database.
- pci memory size – sets the maximum size of the PCI memory pool.

Note You must enable both Java and the PCI Bridge before you can use the PCA/JVM.

See the installation guide for your platform and the *System Administration Guide: Volume 1*.

Configuration options for the PCI Bridge

Do not change any configuration options—directives or arguments—for the PCI Bridge unless instructed to do so by Sybase Technical Support.

Configuration options for the PCA/JVM

You can safely change these arguments for the PCA/JVM:

- pca_jvm_module_path – change this property only if you are using a JRE other than that provided by the installation. If you are, point this property to the JRE to be used by the PCA/JVM.
- pca_jvm_work_dir – add one entry to this argument array for each working directory (trusted directory) that can be configured with a specific permission mask, as needed. See Chapter 8, “File and Network Access Using Java.”
- pca_jvm_netio – enable this argument to enable network I/O. Disable this argument to disable network I/O.
- pca_jvm_dbg_agent_port – enable this argument and set its numeric value to the port number the JVM uses for the debug agent. Your Java debugger must listen on the same port.

- pca_jvm_java_dbg_agent_suspend – enable this argument to start the debug agent in a suspended state. Enabling this argument is useful because it can allow you time to set breakpoints and other options in your Java debugger after it is attached to the running process. See the *Reference Manual: Commands*.

  Note Use pca_jvm_java_dbg_agent_suspend with caution. Enabling pca_jvm_java_dbg_agent_suspend suspends the JVM and all Adaptive Server Java tasks wait until you attach the debugger and instruct the JVM to continue. Sybase recommends that you start the JVM and run a simple Java command to attach the debugger rather than enabling pca_jvm_java_dbg_agent_suspend. Using the Java command allows the JVM to start, and lets you attach the debugger before executing the class that is to be debugged.

Do not change any other directives or arguments for the PCA/JVM without instructions from Sybase Technical Support.

Changing configuration values in a running server

If, with advice from Sybase Technical Support, you want to change the default configuration values, you can use the sp_jreconfig and sp_pciconfig system stored procedures. See "When to change configuration values" on page 19. This section describes how to load the changed configuration values into memory in a running server.

When Adaptive Server starts, it automatically loads the JVM if the server has been configured for Java. The JVM is not initialized, however, until it receives the first Java request. This depends on how frequently Java is used. Changing configuration values before initialization is relatively simple. Changing configuration values after initialization, when the configuration information has been read into in-memory data structures, is more difficult.

You can update configuration information:

- By restarting Adaptive Server
- Before the JVM has been initialized (for the PCA/JVM plug-in only)
- After the JVM has been initialized (for the PCA/JVM plug-in only)

Changing configuration values by restarting Adaptive Server

This is the easiest method of changing configuration information, and it is always available.

Note You must use this method if you are using sp_pciconfig to change configuration values for the PCI Bridge.

1 Use sp_jreconfig or sp_pciconfig to change configuration values.

2 Restart Adaptive Server.

Changing configuration values before the JVM is initialized

Use this method to change configuration values for the PCA/JVM plug-in when Adaptive Server is running, but the JVM is not initialized.

1 Use sp_jreconfig to change configuration values.

2 Load the configuration parameters into memory. Enter:

```
sp_jreconfig "reload_config"
```

You do not need to restart Adaptive Server for the new configuration values to take effect.

Note Changes made with sp_jreconfig "reload_config" take effect only if you have *not yet* initialized the JVM. Using sp_jreconfig modifies only the table values in sybpcidb, and does not affect the current in-memory data structures that were loaded into memory when you started Adaptive Server.

You can safely attempt this method even if you are unsure whether the JVM has been initialized or not. If the JVM has been initialized, the reload_config command fails and an error message displays. There are no negative consequences.

Changing configuration values after the JVM is initialized

If Adaptive Server is running and you have initialized the JVM, the configuration parameters are in memory, and you can change the PCA/JVM plug-in configuration parameters.

The steps you follow for changing the JVM configuration values depend on whether Adaptive Server is configured for threaded or process mode.

- Threaded mode –

  a Use sp_jreconfig to change configuration values.

  b Restart Adaptive Server.

  After Adaptive Server restarts, the JVM is in an uninitiated state until it receives its first Java request.

- Process mode –

  a Use sp_jreconfig to change configuration values.

  b Bring the engine running the JVM offline (in this example, engine number three):

  ```
  sp_engine "offline", 3
  ```

  c Bring the engine running the JVM back online:

  ```
  sp_engine "online", 3
  ```

  Adaptive Server continues to run during this procedure, but Java is not available until you bring the engine running the JVM online. After you bring the engine online, the JVM is again in the uninitiated state until it receives the first Java request.

Restoring default configuration values to *sybpcidb*

The steps for restoring the default configuration values to the sybpcidb configuration values after the JVM has been initialized depend on whether you can restart Adaptive Server, and whether you are using a single- or multiple-engine Adaptive Server.

If you are using a single-engine Adaptive Server:

1 Reinstall the *installpcidb* installation script to reset the sybpcidb configuration table values to their factory defaults. For example:

```
isql -Usa -Psa_password -Sserver_name
-i $SYBASE_ASE/scripts/installpcidb
```

2 Restart Adaptive Server. The default configuration values take effect when the JVM initializes in response to the first Java request.

If you are using a multiple-engine Adaptive Server:

1 Reinstall *installpcidb* to reset the sybpcidb configuration table values to their factory defaults. For example:

```
isql -Usa -Psa_password -Sserver_name
-i $SYBASE_ASE/scripts/installpcidb
```

2 Bring the engine running the JVM offline. For example:

```
sp_engine "offline", 3
```

In this example, the JVM is running on engine “3”.

3 Bring the engine running the JVM back online. For example:

```
sp_engine "online", 3
```

You do not need to restart Adaptive Server for the new configuration values to take effect.

Using monitor tables to display information about the PCI Bridge

You can display information about the PCI Bridge using these monitor tables:

- monPCIBridge – displays general information about the PCI Bridge. For

  For example:

```
select * from monPCIBridge
```

```
Status  ConfiguredSlots  ActiveSlots  ConfiguredPCIMemoryKB  UsedPCIMemoryKB
------------------------------------------------------------------------------
ACTIVE                1            1                  65668             1613
```

- monPCISlots – displays information about the plug-in bound to each slot. For example:

```
select * from monPCISlots

Slot     Status     Modulename    Engine
-----------------------------------------
   1     IN USE        PCA/JVM          0
```

- monPCIEngine – displays engine information for the PCI Bridge and its plug-ins. For example:

```
select * from monPCIEngine

Engine  Status  PLBStatus  NumberofActiveThreads PLBRequest  PLBWakeupRequests
------------------------------------------------------------------------------
     0 PCA ACTIVE  ACTIVE                    10          4                  4
     1 PCA ACTIVE  ACTIVE                     4          0                  0
```

See the *Reference Manual: Tables* for more information.

CHAPTER 3

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.

The Java runtime environment

The Adaptive Server runtime environment for Java requires a third-party JVM, the Sybase PCI, 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 any of the following sources for Java classes:

- The standard Java distribution found in *rt.jar* and classes installed in the "ext" directory under the Java installation directory.

  Note The contents of the ext directory may vary depending on the Java vendor. See the vendor's documentation for detailed information about these classes.

- User-defined classes

Sybase runtime Java classes

The Sybase runtime Java classes are the low-level classes installed to Java-enable a database. They are downloaded automatically when Adaptive Server is installed and are available thereafter from *$SYBASE /$SYBASE_ASE/lib/sybasert.jar* (UNIX) or *%SYBASE%\%SYBASE_ASE%\lib\sybasert.jar* (Windows). Adaptive Server sets the CLASSPATH environment when the JVM starts.

Note If CLASSPATH is set in the operating system environment, Adaptive Server *ignores* that value when the internal JVM starts.

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 native JDBC driver that comes with Adaptive Server supports JDBC versions 1.1 and 1.2, and is compliant with several classes and methods of JDBC version 2.0. See Chapter 9, "Additional Topics," for a complete list of supported and not supported classes and methods.

If your system requires a JDBC driver on the client, you must use jConnect version 6.x or later, which supports JDBC version 2.0.

The JVM

The Adaptive Server Java framework has been designed to work with any standard JVM that supports Java 6 or later. Adaptive Server version 15.0.3 has been certified with the Java 6 version that is included in the *$SYBASE/shared* directory. Classes compiled by earlier versions of Java will continue to run correctly under later versions of Java.

Enabling Java

Note Configure sybpcidb as described in the install guide for your platform before enabling the PCI and Java.

To enable the server and its databases for Java, enter these commands from isql:

```
sp_configure "enable pci", 1
sp_configure "enable java", 1
```

Then, shut down and restart the server. Adaptive Server 15.0.3 and later require that you enable the PCI as a prerequisite to enabling Java.

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.

Installing Java classes in the database

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

See the *Adaptive Server Enterprise Utilities Guide* for detailed information about these utilities. Both utilities perform the same tasks; for simplicity, this document uses UNIX examples.

Using *installjava*

installjava copies an uncompressed 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]
[ -S server_name ]
[ -U user_name ]
[ -P password ]
[ -D database_name ]
[ -I interfaces_file ]
[ -a display_charset ]
[ -J client_charset ]
[ -z language ]
[ -t timeout ]
```

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 Guide*.

Note When you install a JAR file, Application Server copies the file to a temporary table and then installs it from there. If you install a large JAR file, you may need to expand the size of tempdb using the alter database command.

Installing uncompressed JARs

The installjava and instjava tools require an uncompressed jar file.

To install Java classes in a database, save the classes or packages in a JAR file, in uncompressed form. To create an uncompressed JAR file that contains Java classes, use the Java jar cf0 ("zero") 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
```

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

Sybase recommends that you specify a JAR name so that you can better manage your 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.
- You can use extractjava to download the JAR to an operating system file. See “Downloading installed classes and JARs” on page 33.

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 original 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 be associated only 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 34 for more information.

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 reference 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 43.

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 sp_helpjava. The syntax is:

```
sp_helpjava [‘class’ [, name [, 'detail' | , 'depends' ]] |
    ‘jar’ [, name [, 'depends' ] ]]
```

For example, to view detailed information about the Address class, including the version number, log in to isql and enter:

```
sp_helpjava ’class’, Address, detail
```

See “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'
```

See the *Utility Guide* manual for more information.

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 let you remove classes that are used as datatypes for columns and parameters or that are referenced by SQLJ functions or stored procedures. Other classes cannot be checked for usage and may be removed while still referenced in stored procedures. Make sure that you do not remove subclasses or classes that are used as variables or UDF return types.

---

remove java package 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 with the new keyword.

CHAPTER 4

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.

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.

General concepts

This sections provides general Java and Java-SQL identifier information.

Java considerations

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

- Java 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 157.
- Java classes can include classes, fields, and methods that are private, protected, or public.

  Classes, fields and methods that are public can be referenced in SQL. Classes, fields, and methods that are private or protected cannot be referenced in SQL, but they can be referenced in Java, and are subject to normal Java rules.
- Java 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 159 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 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 and altering 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 affects the speed with which Adaptive Server references and updates the fields in these columns.

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

If a Java-SQL column is stored in-row:

- Objects stored in-row are processed more quickly than objects stored off-row.
- An object stored in-row can occupy up to approximately 16K bytes, depending on the page size of the database server and other variables. This includes its entire serialization, not just the values in its fields. A Java object whose runtime representation is more than the 16K limit generates an exception, and the command aborts.

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

- Objects stored off-row are processed more slowly than objects stored in-row.
- An object stored off-row can be of any size—subject to normal limits on text and image columns.
- An off-row column cannot be referenced in a check constraint.

  Similarly, do not reference a table that contains an off-row column in a check constraint. Adaptive Server allows you to include the check constraint when you create or alter the table, but issues a warning message at compile time and ignores the constraint at runtime.
- You cannot include an off-row column in the column list of a select query with select distinct.

- You cannot specify an off-row column in a comparison operator, in a predicate, or in a group by clause.

Partial 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 [ ( size_in_bytes ) ] ] ]...
```

size_in_bytes specifies the maximum size of the in-row column. The value can be as large as 16K bytes. The default value is 255 bytes.

The maximum in-row column size you enter in create table must include the column's entire serialization, not just the values in its fields, plus minimum values for overhead.

To determine an appropriate column size that includes overhead and serialization values, use the datalength system function. datalength allows you to determine the actual size of a representative object you intend to store in the column.

For example:

```
select datalength (new class_name(...))
```

where *class_name* is an installed Java-SQL class.

Partial syntax for alter table is:

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

Note You cannot change the column size of an in-row column using alter column in this Adaptive Server release.

Altering partitioned tables

If a table containing Java columns is partitioned, you cannot alter the table without first dropping the partitions. To change the table schema:

1 Remove the partitions.

2 Use the alter table command.

3 Repartition the table.

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.

Constructor methods are pseudo instance methods. They create instances. Constructor methods have the same name as the class, and have no declared datatype. If you do not include a constructor method in your class definition, a default method is provided by the Java base class 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 ElmStreet', '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')
```

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_add>>street, @zip = home_add>>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 static methods.

Instance methods are generally closely tied to the data encapsulated in a particular instance of their class. Static (class) methods affect the whole class, not a particular instance of the class. Static methods often apply to objects and values from a wide range of classes.

Once you have installed a static method, it is ready for use. A class that contains a static 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 static methods that return a value to the caller as user-defined functions (UDFs).

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

Java methods invoked directly in SQL as UDFs are subject to these limitations:

- If the 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.
- Cross-database invocations of static methods are supported only if you use a class instance as a column value.

Permission to execute any UDF is granted implicitly to public. If the UDF performs SQL queries via JDBC, permission to access the data is checked against the invoker of the UDF. Thus, if user A invokes a UDF that accesses table t1, user A must have select permission on t1 or the query will fail. For a more detailed discussion of security models for Java method invocations, see “Security and permissions” on page 93.

To use Java static methods to return result sets and output parameters, you must enclose the methods in SQL wrappers and invoke them as SQLJ stored procedures or functions. See “Invoking Java methods in Adaptive Server” on page 94 for a comparison of the ways you can invoke Java methods in Adaptive Server.

Sample methods

The sample Address and Address2Line classes have instance methods named toString( ), and the sample Misc class has static 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 55.

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:

```
Unhandled Java method 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

Non-Java clients such as isql cannot receive serialized Java objects from the server. To allow you to view and use the object, Adaptive Server must convert the object to a viewable representation.

To use an actual string value, Adaptive Server 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. You must create your own version of the toString() method so that you can work with the viewable representation of the object.

Note The toString() method in the Java API does not convert the object to a viewable representation. The toString() method you create overrides the toString() method in the Java API.

When you use a toString( ) method, Adaptive Server imposes a limit on the number of bytes returned. Adaptive Server truncates the printable representation of the object to the value of the @@stringsize global variable. The default value of @@stringsize is 50; you can change this value using the set stringsize command. For example:

```
set stringsize 300
```

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_add>>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 deserialize 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 157.

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 and varbinary, where the maximum length value of 16K 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.

In the emps table, the maximum value for the Address and Address2Line classes, street, zip, and line2 fields is 255 bytes (the default value). The Java datatype of these classes is java.String, and they are treated in SQL as varchar(255).

An expression whose datatype is a Java object 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 source code and for Java String data is Unicode. Fields of Java-SQL classes can contain Unicode data.

Note 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.

Subtypes in Java-SQL data

Class subtypes allow you to use subtype substitution and method override, which are 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 versus 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 53 for a description of null values for widening and narrowing conversions.

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.

Java has different rules for the effect of referencing a field or method of a null instance. In Java, if you attempt to reference a field of a null instance, an exception is raised.

For example, suppose that the emps table has the following rows:

```
insert into emps (name, home_addr)
    values ("Al Adams",
    new Address("123 Main", "95321"))

insert into emps (name, home_addr)
    values ("Bob Baker",
    new Address("456 Side", "95123"))

 insert into emps (name, home_addr)
    values ("Carl Carter", null)
```

Consider the following select:

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

If the Java rule were used for the references to "home_addr>>zip," then those references would cause an exception for the "Carl Carter" row, whose "home_addr" column is null. To avoid such an exception, you would need to write such a select as follows:

```
select name,
    case when home_addr is not null then home_addr>>zip
    else null end

from emps
    where case when home_addr is not null
    then home_addr>>zip
else
    null end
in ('95123', '95125', '95128')
```

The SQL convention is therefore used for references to fields and methods of null instances: if the instance is null, then any field or method reference is null. The effect of this SQL rule is to make the above case statement implicit.

However, this SQL 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 D '99123'
    where name D '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, because 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( ) D
    'StreetD234 Stone Road ZIPD 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

The outcome of passing null as a parameter is independent of the actions of the method for which it is an argument, but instead depends 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 because the return type has no null state. A null cannot be passed directly because the method resolution fails without parameter type information.

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 48, 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)” contains 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 runtime type of the expression’s value is a class, subclass, or superclass:

- If the runtime value of the expression (home_addr) is the specified class (Address2Line) or one of its subclasses, 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. If convert returns null, then the field reference itself is null.
- When a convert returns null, then the field reference itself evaluates to null.

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 "Treatment of nulls in Java-SQL data" on page 50, 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 a 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
```

Otherwise, the 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 with a 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.

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 a copy of 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 set 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 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

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. You cannot:

- Reference Java-SQL data items in ordering operations.
- Reference Java-SQL data items in equality operations if they are stored in an off-row column.
- 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 for in-row columns:

- 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.

Evaluation order and Java method calls

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 invoking in the same SQL statement multiple methods on the same Java-SQL object. If at least one of them modifies the object, 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.”

Deterministic Java functions in expressions

Deterministic expressions and functions always return the same result if they are evaluated with the same set of input values. All Java functions in Adaptive Server are deterministic. As a result, if the parameters and input values in an expression involving a Java function do not change, Adaptive Server treats the entire expression as deterministic.

When Adaptive Server encounters a Java function in an expression, Adaptive Server calculates the expression immediately so that the calculation is performed only once and not repeated for each row. This improves performance, but may cause unexpected behavior.

Consider this example:

```
1> create table CaseTest
2> (TestValue varchar(50))
3> go
1> insert into CaseTest values('07')
2> go
(1 row affected)

1> declare @IntArray sybase.cpp.value.client/common.IntArray
2> select @IntArray = new sybase.cpp.value.client.common.IntArray()
3> SELECT CASE
4> WHEN CT.TestValue = '07'
5> THEN @IntArray >> setInt(new java.lang.Integer(10))
6> ELSE @IntArray >> setInt(new java.lang.Integer(11))
7> END
8> FROM CaseTest CT
9> select @IntArray >> getInt(0) as GetObjAfter0
10> select @IntArray >> getInt(1) as GetObjAfter1
11> select @IntArray >> getArraySize() as NumObjectsOnArray
12> go

-----------------------------------------------
sybase.cpp.value.client.common.IntArray@22cc0f30

(1 row affected)
GetObjAfter0
-----------------------------------------------
11

(1 row affected)
NumObjectsOnArray
---------------
2

(1 row affected)
```

You might expect one branch of the case statement to evaluate to true and thus have only one value (10) inserted into the integer array, but because the expressions `setInt(new java.lang.Integer(10))` and `setInt(new java.lang.Integer(11))` are deterministic, Adaptive Server "precalculates" the result, and populates the array with both values.

You can make expressions nondeterministic by adding a reference to columns so that Adaptive Server does not know that the expressions produce the same result for each execution. For example, make these changes to the Transact-SQL statements in the example:

```
1> declare @IntArray Sybase.cpp.value.client.common.IntArray
2> select @IntArray = new sybase.cpp.value.client.common.IntArray()
3> SELECT CASE
4> WHEN CT.TestValue = '07'
5> THEN @IntArray >> setInt(new java.lang.Integer(10 +
convert(int,CT.TestValue) - convert(int,CT.TestValue)))
6> ELSE @IntArray >> setInt(new java.lang.Integer(11 +
convert(int,CT.TestValue) - convert(int,CT.TestValue)))
7> END
8> FROM CaseTest CT
9> select @IntArray >> getInt(0) as GetObjAfter0
10> select @IntArray >> getInt(1) as GetObjAfter1
11> select @IntArray >> getArraySize() as NumObjectsOnArray
12> go
```

By including the column references in the THEN and ELSE portions of the case statement, the optimizer no longer treats the statements as constants and does not precalculate the Java insert statement.

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 of a Java class in SQL by qualifying the static variable with an instance of the class. For example:

```
declare @a Address
select @a>>recommendedLimit
```

If you don't have an instance of the class, you can use the following technique:

```
select convert(Address, null)>>recommendedLimit
```

The expression "(convert(null, Address))" converts a null value to an Address type; that is, it generates a null Address instance, which you can then qualify with the static variable name. You cannot reference a static variable of a Java class in SQL by qualifying the static variable with the class name. For example, the following are both incorrect:

```
select Address.recommendedLimit

select Address>>recommendedLimit
```

Values assigned to nonfinal static variables are accessible only within the current session.

Changes for static variables for Adaptive Server 15.0.3 and later

In Adaptive Server 15.0.2 and earlier, each task was assigned its own internal JVM. Each JVM was associated with a unique set of ClassLoaders. As a result, class variables were available only to a single Adaptive Server task.

With Adaptive Server 15.0.3 and later, and the introduction of the PCA/JVM, Adaptive Server uses a separate JVM thread for each Adaptive Server task within the same JVM. All user classes are loaded by ClassLoaders associated only with the specific Adaptive Server task executing the particular Java method. Because ClassLoaders associated with user classes are not shared across Adaptive Server tasks, user classes are not considered the same. Therefore, class variables from user classes are not visible across Adaptive Server tasks.

However, class variables from classes loaded by the system ClassLoader are visible across all Adaptive Server tasks because all user ClassLoaders share the system ClassLoader as a parent. This is true for all standard JVMs. Class variables in these classes do not endanger functionality or security when they are used across multiple tasks.

Changes for static variables for the Cluster Edition

In the Cluster Edition, Adaptive Server handles class variables from user classes and classes loaded by the system ClassLoader as described in “Changes for static variables for Adaptive Server 15.0.3 and later” on page 63: however, each node has a separate, unrelated PCA/JVM instance running. If you set a class variable on one node, that value is not automatically changed on all other nodes in the cluster. Because an Adaptive Server task can run across multiple nodes, if user classes rely on class variables, you must explicitly set that same class variable on all nodes.

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, it is 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 objects stored in table columns 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 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. If the class in the source database is not compatible with the class in the target database, then the Java exception InvalidClassException is raised.

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 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.

Java classes

This section shows the simple Java classes that this chapter uses to illustrate Java in Adaptive Server.

This is the Address class:

```
//
// Copyright (c) 2005
// Sybase, Inc
// Dublin, CA 94568
// 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) 2005
// Sybase, Inc
// Dublin, CA 94568
// 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) 2005
// Sybase, Inc
// Dublin, CA 94568
// 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 5

Data Access Using JDBC

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

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 native 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( ) static 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.

When you use client-side or server-side JDBC, you call the Drivermanager.getConnection() method to establish a connection to the server.

- For client-side JDBC, you use the Sybase jConnect JDBC driver, and call the Drivermanager.getConnection() method with the identification of the server. This establishes a connection to the designated server.

- For server-side JDBC, you use the Adaptive Server native JDBC driver, and call the Drivermanager.getConnection() method with one of the following values:
  - jdbc:default:connection
  - jdbc:sybase:ase
  - jdbc:default
  - empty string

  This establishes a connection to the current server. Only the first call to the getConnection() method creates a new connection to the current server. Subsequent calls return a wrapper of that connection with all connection properties unchanged.

You can write JDBC classes to run at both the client and the server by using a conditional statement to set the URL.

Permissions

- *Java execution 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 in Java methods, and there is no need to qualify the name of a class with the name of its owner.
- *SQL 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 granted permission by the database owner.

Using JDBC to access data

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 84.

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

Overview of the JDBCExamples class

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 native driver does not support create procedure and drop procedure statements.

JDBCExamples static 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, inoutproc:

```
create procedure inoutproc @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 inoutproc, which has both input parameters and output parameters of datatypes java.lang.String and Address.

JDBCExamples operates only on the xmp table and inoutproc 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, using the dsedit tool. *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 inoutproc 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 inoutproc, 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 inoutproc with client-side calls of the main( ) method before executing these examples. Refer to "Overview of the JDBCExamples class" on page 76.

After creating xmp and inoutproc, 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 *updater( )* method

The updater( ) 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 (?). updater( ) 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, updater( ) executes the update statement.

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

Executing a *select* statement: the *selecter( )* method

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

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

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

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

selecter( ) then executes the select statement:

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

Note For SQL statements that return no results, use doSQL( ) and updater( ). 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.

selecter( ) 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.

selecter( ) 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.

Executing in batch mode

If you want to execute a batch of SQL statements, make sure that you use the execute( ) method. If you use executeQuery( ) for batch mode:

- If the batch operation does not return a result set (contains no select statements), the batch executes without error.
- If the batch operation returns one result set, all statements after the statement that returns the result are ignored. If getXXX( ) is called to get an output parameter, the remaining statements execute and the current result set is closed.
- If the batch operation returns more than one result set, an exception is raised and the operation aborts.

Using execute( ) ensures that the complete batch executes for all cases.

Calling a SQL stored procedure: the *caller( )* method

The caller( ) method calls the stored procedure inoutproc:

```
create proc inoutproc @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*). caller( ) 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 inoutproc procedure illustrates how to supply input and output parameters in a JDBC call.

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

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

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

caller( ) supplies values for the input parameters using JDBC setInt( ), setString( ), and setObject( ) methods that were used in the doSQL( ), updatAction( ), and selecter( ) 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, caller( ) specifies the datatypes expected of the output parameters using the JDBC registerOutParameter( ) method:

```
cs.registerOutParameter(4, java.sql.Types.VARCHAR);
cs.registerOutParameter(5, java.sql.Types.JAVA_OBJECT);
```

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

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

Error handling in the native JDBC driver

Sybase supports and implements all methods from the java.sql.SQLException and java.sql.SQLWarning classes. SQLException provides information on database access errors. SQLWarning extends SQLException and provides information on database access warnings.

Errors raised by Adaptive Server are numbered according to severity. Lower numbers are less severe; higher numbers are more severe. Errors are grouped according to severity:

- Warnings (EX_INFO: severity 10) – are converted to SQLWarnings.
- Exceptions (severity 11 to18) – are converted to SQLExceptions.
- Fatal errors (severity 19 to 24) – are converted to fatal SQLExceptions.

SQLExceptions can be raised through JDBC, Adaptive Server, or the native JDBC driver. Raising a SqlException aborts the JDBC query that caused the error. Subsequent system behavior differs depending on where the error is caught:

- *If the error is caught in Java* – a “try” block and subsequent “catch” block process the error.

Adaptive Server provides several extended JDBC driver-specific SQLException error messages. All are EX_USER (severity 16) and can always be caught. There are no driver-specific SQLWarning messages.

- *If the error is not caught in Java* – the Java VM returns control to Adaptive Server, Adaptive Server catches the error, and an unhandled SQLException error is raised.

  The raiserror command is used typically with stored procedures to raise an error and to print a user-defined error message. When a stored procedure that calls the raiserror command is executed via JDBC, the error is treated as an internal error of severity EX_USER, and a nonfatal SQLException is raised.

Note You cannot access extended error data using the raiserror command; the with errordata clause is not implemented for SQLException.

If an error causes a transaction to abort, the outcome depends on the transaction context in which the Java method is invoked:

- *If the transaction contains multiple statements* – the transaction aborts and control returns to the server, which rolls back the entire transaction. The JDBC driver ceases to process queries until control returns from the server.
- *If the transaction contains a single statement* – the transaction aborts, the SQL statement it contains rolls back, and the JDBC driver continues to process queries.

The following scenarios illustrate the different outcomes. Consider a Java method jdbcTests.Errorexample() that contains these statements:

```
stmt.executeUpdate("delete from parts where partno = 0"); Q2
stmt.executeQuery("select 1/0"); Q3
stmt.executeUpdate("delete from parts where partno = 10"); Q4
```

A transaction containing multiple statements includes these SQL commands:

```
begin transaction
delete from parts where partno = 8 Q1
select JDBCTests.Errorexample()
```

In this case, these actions result from an aborted transaction:

- A divide-by-zero exception is raised in Q3.
- Changes from Q1 and Q2 are rolled back.
- The entire transaction aborts.

A transaction containing a single statement includes these SQL commands:

```
set chained off
delete from parts where partno = 8 Q1
select JDBCTests.Errorexample()
```

In this case:

- A divide-by-zero exception is raised in Q3.
- Changes from Q1 and Q2 are not rolled back
- The exception is caught in “catch” and “try” blocks in JDBCTests.Errorexample.
- The deletion specified in Q4 does not execute because it is handled in the same “try” and “catch” blocks as Q3.
- JDBC queries outside of the current “try” and “catch” blocks can be executed.

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 serverMain 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 serverMain( ) method

```
// A JDBCExamples method equivalent to 'main',
// to be called from SQL or Java in the server

    public static String serverMain(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 inoutproc @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 inoutproc ";
          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 += updater(con);
        } else if (action.equals("select")) {
             workString += selecter(con);
        } else if (action.equals("call")) {
             workString += caller(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 updater( ) method

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

    public static String updater(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 selecter( ) 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 selecter(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 caller( ) 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 caller(Connection con)
           throws Exception {
        CallableStatement cs  = con.prepareCall("{call inoutproc
          (?, ?, ?, ?, ?)}");
        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, java.sql.Types.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 6

SQLJ Functions and Stored Procedures

This chapter describes how to wrap Java methods in SQL names and use them as Adaptive Server functions and stored procedures.

Overview

You can enclose Java static methods in SQL wrappers and use them exactly as you would Transact-SQL stored procedures or built-in functions. This functionality:

- Allows Java methods to return output parameters and result sets to the calling environment.
- Complies with Part 1 of the ANSI SQLJ standard specification.
- Allows you to take advantage of traditional SQL syntax, metadata, and permission capabilities.
- Allows you to use existing Java methods as SQLJ procedures and functions on the server, on the client, and on any SQLJ-compliant, third-party database.

❖ **Creating a SQLJ stored procedure or function**

Perform these steps to create and execute a SQLJ stored procedure or function.

1 Create and compile the Java method. Install the method class in the database using the installjava utility.

Refer to Chapter 3, “Preparing for and Maintaining Java in the Database,” for information on creating, compiling, and installing Java methods in Adaptive Server.

2 Using the SQLJ create procedure or create function statement, define a SQL name for the method.

3 Execute the procedure or function. The examples in this chapter use JDBC method calls or isql. You can also execute the method using Embedded SQL or ODBC.

Compliance with SQLJ Part 1 specifications

Adaptive Server SQLJ stored procedures and functions comply with SQLJ Part 1 of the standard specifications for using Java with SQL. See “Standards” on page 7 for a description of the SQLJ standards.

Adaptive Server supports most features described in the SQLJ Part 1 specification; however, there are some differences. Unsupported features are listed in Table 6-3 on page 119; partially supported features are listed in Table 6-4 on page 119. Sybase-defined features—those not defined by the standard but left to the implementation—are listed in Table 6-5 on page 119.

In those instances where Sybase proprietary implementation differs from the SQLJ specifications, Sybase supports the SQLJ standard. For example, non-Java Sybase SQL stored procedures support two parameter modes: in and inout. The SQLJ standard supports three parameter modes: in, out, and inout. The Sybase syntax for creating SQLJ stored procedures supports all three parameter modes.

General issues

This section describes general issues and constraints that apply to SQLJ functions and stored procedures.

Security and permissions

Sybase provides different security models for SQLJ stored procedures and SQLJ functions.

SQLJ functions and user-defined functions (UDFs) (see “Invoking Java methods in SQL” on page 41) use the same security model. Permission to execute any UDF or SQLJ function is granted implicitly to public. If the function performs SQL queries via JDBC, permission to access the data is checked against the invoker of the function. Thus, if user A invokes a function that accesses table t1, user A must have select permission on t1 or the query fails.

SQLJ stored procedures use the same security model as Transact-SQL stored procedures. The user must be granted explicit permission to execute a SQLJ or Transact-SQL stored procedure. If a SQLJ procedure performs SQL queries via JDBC, implicit permission grant support is applied. This security model allows the owner of the stored procedure, if the owner owns all SQL objects referenced by the procedure, to grant execute permission on the procedure to another user. The user who has execute permission can execute all SQL queries in the stored procedure, even if the user does not have permission to access those objects.

In general, after the JVM is configured and running, any user able to access Java classes from the database can run them. However, the following operations are restricted:

- Thread operations except those required to create and join
- System operations that affect the server such as exit( ) and abort( )
- Changes to the class loader hierarchy
- Override of the installed SecurityManager

For a more detailed description of security for stored procedures, see the *Security Administration Guide*.

SQLJ Examples

The examples used in this chapter assume a SQL table called sales_emps with these columns:

- name – the employee’s name
- id – the employee’s identification number
- state – the state in which the employee is located

- sales – amount of the employee's sales
- jobcode – the employee's job code

The table definition is:

```
create table sales_emps
    (name varchar(50), id char(5),
    state char(20), sales decimal (6,2),
    jobcode integer null)
```

The example class is SQLJExamples, and the methods are:

- region() – maps a U.S. state code to a region number. The method does not use SQL.
- correctStates() – performs a SQL update command to correct the spelling of state codes. Old and new spellings are specified by input parameters.
- bestTwoEmps() – determines the top two employees by their sales records and returns those values as output parameters.
- SQLJExamplesorderedEmps() – creates a SQL result set consisting of selected employee rows ordered by values in the sales column, and returns the result set to the client.
- job() – returns a string value corresponding to an integer job code value.

See "SQLJExamples class" on page 120 for the text of each method.

Invoking Java methods in Adaptive Server

You can invoke Java methods in two different ways in Adaptive Server:

- Invoke Java methods directly in SQL. Directions for invoking methods in this way are presented in Chapter 4, "Using Java Classes in SQL."
- Invoke Java methods indirectly using SQLJ stored procedures and functions that provide Transact-SQL aliases for the method name. This chapter describes invoking Java methods in this way.

Whichever way you choose, you must first create your Java methods and install them in the Adaptive Server database using the installjava utility. See Chapter 3, "Preparing for and Maintaining Java in the Database," for more information.

Invoking Java methods directly with their Java names

You can invoke Java methods in SQL by referencing them with their fully qualified Java names. Reference instances for instance methods, and either instances or classes for static methods.

You can use static methods as user-defined functions (UDFs) that return a value to the calling environment. You can use a Java static method as a UDF in stored procedures, triggers, where clauses, select statements, or anywhere that you can use a built-in SQL function.

When you call a Java method using its name, you cannot use methods that return output parameters or result sets to the calling environment. A method can manipulate the data it receives from a JDBC connection, but the method can only return the single return value declared in its definition to the calling environment.

You cannot use cross-database invocations of UDF functions.

See Chapter 4, “Using Java Classes in SQL,” for information about using Java methods in this way.

Invoking Java methods indirectly using SQLJ

You can invoke Java methods as SQLJ functions or stored procedures. By wrapping the Java method in a SQL wrapper, you take advantage of these capabilities:

- You can use SQLJ stored procedures to return result sets and output parameters to the calling environment.
- You can take advantage of SQL metadata capabilities. For example, you can view a list of all stored procedures or functions in the database.
- SQLJ provides a SQL name for a method, which allows you to protect the method invocation with standard SQL permissions.
- Sybase SQLJ conforms to the recognized SQLJ Part 1 standard, which allows you to use Sybase SQLJ procedures and functions in conforming non-Sybase environments.
- You can invoke SQLJ functions and SQLJ stored procedures across databases.
- Because Adaptive Server checks datatype mapping when the SQLJ routine is created, you need not be concerned with datatype mapping when executing the routines.

You must reference static methods in a SQLJ routine; you cannot reference instance methods.

This chapter describes how you can use Java methods as SQLJ stored procedures and functions.

Using Sybase Central to manage SQLJ functions and procedures

You can manage SQLJ functions and procedures from the command line using isql and from the Adaptive Server plug-in to Sybase Central. From the Adaptive Server plug-in you can:

- Create a SQLJ function or procedure
- Execute a SQLJ function or procedure
- View and modify the properties of a SQLJ function or procedure
- Delete a SQLJ function or procedure
- View the dependencies of a SQLJ function or procedure
- Create permissions for a SQLJ procedure

The following procedures describes how to create and view the properties of a SQLJ routine. You can view dependencies and create and view permissions from the routine's property sheet.

❖ **Creating a SQLJ function/procedure**

First, create and compile the Java method. Install the method class in the database using installjava. Then follow these steps:

1 Start the Adaptive Server plug-in and connect to Adaptive Server.

2 Double-click on the database in which you want to create the routine.

3 Open the SQLJ Procedures/SQLJ Functions folder.

4 Double-click the Add new Java Stored Procedure/Function icon.

5 Use the Add new Java Stored Procedure/Function wizard to create the SQLJ procedure or function.

When you have finished using the wizard, the Adaptive Server plug-in displays the SQLJ routine you have created in an edit screen, where you can modify the routine and execute it.

❖ **To view the properties of a SQLJ function or procedure**

1 Start the Adaptive Server plug-in and connect to Adaptive Server.

2 Double-click on the database in which the routine is stored.

3 Open the SQLJ Procedures/SQLJ Functions folder.

4 Highlight a function or procedure icon.

5 Select File | Properties.

SQLJ user-defined functions

The create function command specifies a SQLJ function name and signature for a Java method. You can use SQLJ functions to read and modify SQL and to return a value described by the referenced method.

The SQLJ syntax for create function is:

```
create function [owner].sql_function_name
       ([sql_parameter_name sql_datatype
          [( length)| (precision[, scale])]
       [, sql_parameter_name sql_datatype
          [( length )  | ( precision[, scale]) ]]
       ...])
   returns sql_datatype
      [( length)| (precision[, scale])]
   [modifies sql data]
   [returns null on null input |
      called on null input]
   [deterministic | not deterministic]
   [exportable]
   language java
   parameter style java
   external name 'java_method_name
      [([java_datatype[ {, java_datatype }
      ...])]'
```

When creating a SQLJ function:

- The **SQL function signature** is the SQL datatype *sql_datatype* of each function parameter.
- To comply with the ANSI standard, do not include an @ sign before parameter names.

Sybase adds an @ sign internally to support parameter name binding. You will see the @ sign when using sp_help to print out information about the SQLJ stored procedure.

- When creating a SQLJ function, you must include the parentheses that surround the *sql_parameter_name* and *sql_datatype* information—even if you do not include that information.

  For example:

  ```
  create function sqlj_fc()
     language java
     parameter style java
  external name 'SQLJExamples.method'
  ```

- The modifies sql data clause specifies that the method invokes SQL operations and reads and modifies SQL data. This is the default value. You do not need to include it except for syntactic compatibility with the SQLJ Part 1 standard.

- es returns null on null input and called on null input specify how Adaptive Server handles null arguments of a function call. returns null on null input specifies that if the value of any argument is null at runtime, the return value of the function is set to null and the function body is not invoked. called on null input is the default. It specifies that the function is invoked regardless of null argument values.

  Function calls and null argument values are described in detail in "Handling nulls in the function call" on page 101.

- You can include the deterministic or not deterministic keywords, but Adaptive Server does not use them. They are included for syntactic compatibility with the SQLJ Part 1 standard.

- Clauses exportable keyword specifies that the function is to run on a remote server using Sybase OmniConnect™ capabilities. Both the function and the method on which it is based must be installed on the remote server.

- Clauses language java and parameter style java specify that the referenced method is written in Java and that the parameters are Java parameters. You *must* include these phrases when creating a SQLJ function.

- The external name clause specifies that the routine is not written in SQL and identifies the Java method, class and, package name (if any).

- The Java method signature specifies the Java datatype *java_datatype* of each method parameter. The Java method signature is optional. If it is not specified, Adaptive Server infers the Java method signature from the SQL function signature.

  Sybase recommends that you include the method signature as this practice handles all datatype translations. See "Mapping Java and SQL datatypes" on page 113.

- You can define different SQL names for the same Java method using create function and then use them in the same way.

Writing the Java method

Before you can create a SQLJ function, you must write the Java method that it references, compile the method class, and install it in the database.

In this example, SQLJExamples.region() maps a state code to a region number and returns that number to the user.

```
public static int region(String s)
       throws SQLException {
   s = s.trim();
   if (s.equals "MN") || s.equals("VT") ||
      s.equals("NH") ) return 1;
   if (s.equals("FL") || s.equals("GA") ||
      s.equals("AL") ) return 2;
   if (s.equals("CA") || s.equals("AZ") ||
      s.equals("NV") ) return 3;
   else throw new SQLException
      ("Invalid state code", "X2001");
}
```

Creating the SQLJ function

After writing and installing the method, you can create the SQLJ function. For example:

```
create function region_of(state char(20))
       returns integer
language java parameter style java
external name
       'SQLJExamples.region(java.lang.String)'
```

The SQLJ create function statement specifies an input parameter `(state char(20))` and an integer return value. The SQL function signature is char(20). The Java method signature is java.lang.String.

Calling the function

You can call a SQLJ function directly, as if it were a built-in function. For example:

```
select name, region_of(state) as region
    from sales_emps
where region_of(state)=3
```

Note The search sequence for functions in Adaptive Server is:

1 Built-in functions

2 SQLJ functions

3 Java-SQL functions that are called directly

Handling null argument values

Java class datatypes and Java primitive datatypes handle null argument values in different ways.

- **Java object datatypes** that are classes—such as java.lang.Integer, java.lang.String, java.lang.byte[], and java.*sql*.Timestamp—can hold both actual values and null reference values.
- **Java primitive datatypes**—such as boolean, byte, short, and int—have no representation for a null value. They can hold only non-null values.

When a Java method is invoked that causes a SQL null value to be passed as an argument to a Java parameter whose datatype is a Java class, it is passed as a Java null reference value.When a SQL null value is passed as an argument to a Java parameter of a Java primitive datatype, however, an exception is raised because the Java primitive datatype has no representation for a null value.

Typically, you will write Java methods that specify Java parameter datatypes that are classes. In this case, nulls are handled without raising an exception. If you choose to write Java functions that use Java parameters that cannot handle null values, you can either:

- Include the returns null on null input clause when you create the SQLJ function, or
- Invoke the SQLJ function using a case or other conditional expression to test for null values and call the SQLJ function only for the non-null values.

You can handle expected nulls when you create the SQLJ function or when you call it. The following sections describe both scenarios, and reference this method:

```
public static String job(int jc)
```

```
        throws SQLException {
if (jc==1) return "Admin";
else if (jc==2) return "Sales";
else if (jc==3) return "Clerk";
else return "unknown jobcode";
}
```

Handling nulls when creating the function

If null values are expected, you can include the returns null on null input clause when you create the function. For example:

```
create function job_of(jc integer)
    returns varchar(20)
returns null on null input
language java parameter style java
external name 'SQLJExamples.job(int)'
```

You can then call job_of in this way:

```
select name, job_of(jobcode)
    from sales_emp
where job_of(jobcode) <> "Admin"
```

When the SQL system evaluates the call job_of(jobcode) for a row of sales_emps in which the jobcode column is null, the value of the call is set to null without actually calling the Java method SQLJExamples.job. For rows with non-null values of the jobcode column, the call is performed normally.

Thus, when a SQLJ function created using the returns null on null input clause encounters a null argument, the result of the function call is set to null and the function is not invoked.

Note If you include the returns null on null input clause when creating a SQLJ function, the returns null on null input clause applies to *all* function parameters, including nullable parameters.

If you include the called on null input clause (the default), null arguments for non-nullable parameters generates an exception.

Handling nulls in the function call

You can use a conditional function call to handle null values for non-nullable parameters. The following example uses a case expression:

```
select name,
    case when jobcode is not null
       then job_of(jobcode)
       else null end
from sales_emps where
    case when jobcode is not null
       then job_of(jobcode)
       else null end <> "Admin"
```

In this example, we assume that the function job_of was created using the default clause called on null input.

Deleting a SQLJ function name

You can delete the SQLJ function name for a Java method using the drop function command. For example, enter:

```
drop function region_of
```

which deletes the region_of function name and its reference to the SQLJExamples.region method. drop function does not affect the referenced Java method or class.

See the *Reference Manual: Building Blocks* for complete syntax and usage information.

SQLJ stored procedures

Using Java-SQL capabilities, you can install Java classes in the database and then invoke those methods from a client or from within the SQL system. You can also invoke Java static (class) methods in another way—as SQLJ stored procedures.

SQLJ stored procedures:

- Can return result sets and/or output parameters to the client
- Behave exactly as Transact-SQL stored procedures when executed
- Can be called from the client using ODBC, isql, or JDBC
- Can be called within the server from other stored procedures or native Adaptive Server JDBC

The end user need not know whether the procedure being called is a SQLJ stored procedure or a Transact-SQL stored procedure. They are both invoked in the same way.

The SQLJ syntax for create procedure is:

```
create procedure [owner.]sql_procedure_name
    ([[ in | out | inout ] sql_parameter_name
       sql_datatype [( length) |
       (precision[, scale])]
    [, [ in | out | inout ] sql_parameter_name
       sql_datatype [( length) |
       (precision[, scale]) ]]
    ...])
    [modifies sql data]
    [dynamic result sets integer]
    [deterministic | not deterministic]
    language java
    parameter style java
    external name 'java_method_name
       [([java_datatype[, java_datatype
    ...]])]'
```

Note To comply with the ANSI standard, the SQLJ create procedure command syntax is different from syntax used to create Sybase Transact-SQL stored procedures.

Refer to the *Reference Manual: Commands* for a detailed description of each keyword and option in this command.

When creating SQLJ stored procedures:

- The **SQL procedure signature** is the SQL datatype *sql_datatype* of each procedure parameter.
- When creating a SQLJ stored procedure, do not include an @ sign before parameter names. This practise is compliant with the ANSI standard.

  Sybase adds an @ sign internally to support parameter name binding. You will see the @ sign when using sp_help to print out information about the SQLJ stored procedure.

- When creating a SQLJ stored procedure, you must include the parentheses that surround the *sql_parameter_name* and *sql_datatype* information—even if you do not include that information.

  For example:

  ```
  create procedure sqlj_sproc ()
     language java
     parameter style java
  external name "SQLJExamples.method1"
  ```

- You can include the keywords modifies *sql* data to indicate that the method invokes SQL operations and reads and modifies SQL data. This is the default value.
- You must include the dynamic result sets *integer* option when result sets are to be returned to the calling environment. Use the *integer* variable to specify the maximum number of result sets expected.
- You can include the keywords deterministic or not deterministic for compatibility with the SQLJ standard. However, Adaptive Server does not make use of this option.
- You must include the language java parameter and style java keywords, which tell Adaptive Server that the external routine is written in Java and the runtime conventions for arguments passed to the external routine are Java conventions.
- The external name clause indicates that the external routine is written in Java and identifies the Java method, class, and package name (if any).
- The Java method signature specifies the Java datatype *java_datatype* of each method parameter. The Java method signature is optional. If one is not specified, Adaptive Server infers one from the SQL procedure signature.

  Sybase recommends that you include the method signature as this practice handles all datatype translations. See “Mapping Java and SQL datatypes” on page 113 for more information.
- You can define different SQL names for the same Java method using create procedure and then use them in the same way.

Modifying SQL data

You can use a SQLJ stored procedure to modify information in the database. The method referenced by the SQLJ procedure must be either:

- A method of type void, or
- A method with an int return type (incorporation of the int return type is a Sybase extension of the SQLJ standard).

Writing the Java method

The method SQLJExamples.correctStates() performs a SQL update statement to correct the spelling of state codes. Input parameters specify the old and new spellings. correctStates() is a void method; no value is returned to the caller.

```
public static void correctStates(String oldSpelling,
String newSpelling) throws SQLException {

    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
       Class.forName("sybase.asejdbc.ASEDriver");
       conn = DriverManager.getConnection
          ("jdbc:default:connection");
    }
    catch (Exception e) {
       System.err.println(e.getMessage() +
          ":error in connection");
    }
    try {
       pstmt = conn.prepareStatement
          ("UPDATE sales_emps SET state = ?
          WHERE state = ?");
       pstmt.set.String(1, newSpelling);
       pstmt.set.String(2, oldSpelling);
       pstmt.executeUpdate();
    }
    catch (SQLException e) {
       System.err.println("SQLException: "+
       e.getErrorCode() + e.getMessage());
    }
    return;
}
```

Creating the stored procedure

Before you can call a Java method with a SQL name, you must create the SQL name for it using the SQLJ create procedure command. The modifies sql data clause is optional.

```
create procedure correct_states(old char(20),
```

```
    not_old char(20))
modifies sql data
language java parameter style java
external name
    'SQLJExamples.correctStates
    (java.lang.String, java.lang.String)'
```

The correct_states procedure has a SQL procedure signature of char(20), char(20). The Java method signature is java.lang.String, java.lang.String.

Calling the stored procedure

You can execute the SQLJ procedure exactly as you would a Transact-SQL procedure. In this example, the procedure executes from isql:

```
execute correct_states 'GEO', 'GA'
```

Using input and output parameters

Java methods do not support output parameters. When you wrap a Java method in SQL, however, you can take advantage of Sybase SQLJ capabilities that allow input, output, and input/output parameters for SQLJ stored procedures.

When you create a SQLJ procedure, you identify the mode for each parameter as in, out, or inout.

- For input parameters, use the in keyword to qualify the parameter. in is the default; Adaptive Server assumes an input parameter if you do not enter a parameter mode.
- For output parameters, use the out keyword.
- For parameters that can pass values both to and from the referenced Java method, use the inout keyword.

Note You create Transact-SQL stored procedures using only the in and out keywords. The out keyword corresponds to the SQLJ inout keyword. See the create procedure reference pages in the *Reference Manual: Commands* for more information.

To create a SQLJ stored procedure that defines output parameters, you must:

- Define the output parameter(s) using either the out or inout option when you create the SQLJ stored procedure.
- Declare those parameters as Java arrays in the Java method. SQLJ uses arrays as containers for the method's output parameter values.

For example, if you want an Integer parameter to return a value to the caller, you must specify the parameter type as Integer[] (an array of Integer) in the method.

The array object for an out or inout parameter is created implicitly by the system. It has a single element. The input value (if any) is placed in the first (and only) element of the array before the Java method is called. When the Java method returns, the first element is removed and assigned to the output variable. Typically, this element will be assigned a new value by the called method.

The following examples illustrate the use of output parameters using a Java method bestTwoEmps() and a stored procedure best2 that references that method.

Writing the Java method

The SQLJExamples.bestTwoEmps() method returns the name, ID, region, and sales of the two employees with the highest sales performance records. The first eight parameters are output parameters requiring a containing array. The ninth parameter is an input parameter and does not require an array.

```
public static void bestTwoEmps(String[] n1,
    String[] id1, int[] r1,
    BigDecimal[] s1, String[] n2,
    String[] id2, int[] r2, BigDecimal[] s2,
    int regionParm) throws SQLException {

n1[0] = "****";
id1[0] = "";
r1[0] = 0;
s1[0] = new BigDecimal(0):
n2[0] = "****",
id2[0] = "";
r2[0] = 0;
s2[0] = new BigDecimal(0);

try {
    Connection conn = DriverManager.getConnection
        ("jdbc:default:connection");
    java.sql.PreparedStatement stmt =
        conn.prepareStatement("SELECT name, id,"
        + "region_of(state) as region, sales FROM"
        + "sales_emps WHERE"
        + "region_of(state)>? AND"
        + "sales IS NOT NULL ORDER BY sales DESC");
    stmt.setInteger(1, regionParm);
    ResultSet r = stmt.executeQuery();
```

```
            if(r.next()) {
                n1[0] = r.getString("name");
                id1[0] = r.getString("id");
                r1[0] = r.getInt("region");
                s1[0] = r.getBigDecimaL("sales");
            }
            else return;

            if(r.next()) {
                n2[0] = r.getString("name");
                id2[0] = r.getString("id");
                r2[0] = r.getInt("region");
                s2[0] = r.getBigDecimal("sales");
            }
            else return;
        }
        catch (SQLException e) {
            System.err.println("SQLException: "+
                e.getErrorCode() + e.getMessage());
            }
        }
```

Creating the SQLJ procedure

Create a SQL name for the bestTwoEmps method. The first eight parameters are output parameters; the ninth is an input parameter.

```
create procedure best2
    (out n1 varchar(50), out id1 varchar(5),
    out s1 decimal(6,2), out r1 integer,
    out n2 varchar(50), out id2 varchar(50),
    out r2 integer, out s2 decimal(6,2),
    in region integer)
    language java
    parameter style java
    external name
        'SQLJExamples.bestTwoEmps (java.lang.String,
        java.lang.String, int, java.math.BigDecimal,
        java.lang.String, java.lang.String, int,
        java.math.BigDecimal, int)'
```

The SQL procedure signature for best2 is: varchar(20), varchar(5), decimal (6,2) and so on. The Java method signature is String, String, int, BigDecimal and so on.

Calling the procedure

After the method is installed in the database and the SQLJ procedure referencing the method has been created, you can call the SQLJ procedure.

At runtime, the SQL system:

1 Creates the needed arrays for the out and inout parameters when the SQLJ procedure is called.

2 Copies the contents of the parameter arrays into the out and inout target variables when returning from the SQLJ procedure.

The following example calls the best2 procedure from isql. The value for the region input parameter specifies the region number.

```
declare @n1 varchar(50), @id1 varchar(5),
@s1 decimal (6,2), @r1 integer, @n2 varchar(50),
@id2 varchar(50), @r2 integer, @s2 decimal(6,2),
@region integer
select @region = 3
execute best2 @n1 out, @id1 out, @s1 out, @r1 out,
@n2 out, @id2 out, @r2 out, @s2 out, @region
```

Note Adaptive Server calls SQLJ stored procedures exactly as it calls Transact-SQL stored procedures. Thus, when using isql or any other non-Java client, you must precede parameter names by the @ sign.

Returning result sets

A SQL result set is a sequence of SQL rows that is delivered to the calling environment.

When a Transact-SQL stored procedure returns one or more results sets, those result sets are implicit output from the procedure call. That is, they are not declared as explicit parameters or return values.

Java methods can return Java result set objects, but they do so as explicitly declared method values.

To return a SQL-style result set from a Java method, you must first wrap the Java method in a SQLJ stored procedure. When you call the method as a SQLJ stored procedure, the result sets, which are returned by the Java method as Java result set objects, are transformed by the server to SQL result sets.

When writing the Java method to be invoked as a SQLJ procedure that returns a SQL-style result set, you must specify an additional parameter to the method for each result set that the method can return. Each such parameter is a single-element array of the Java ResultSet class.

This section describes the basic process of writing a method, creating the SQLJ stored procedure, and calling the method. See “Specifying Java method signatures explicitly or implicitly” on page 115 for more information about returning result sets.

Writing the Java method

The following method, SQLJExamples.orderedEmps, invokes SQL, includes a ResultSet parameter, and uses JDBC calls for securing a connection and opening a statement.

```
public static void orderedEmps
       (int regionParm, ResultSet[] rs) throws
       SQLException {

   Connection conn = null;
   PreparedStatement pstmt = null;

   try {
      Class.forName
         ("sybase.asejdbc.ASEDriver");
      Connection conn =
         DriverManager.getConnection
         ("jdbc:default:connection");
   }
   catch (Exception e) {
      System.err.println(e.getMessage()
         + ":error in connection");
   }

   try {
      java.sql.PreparedStatement
         stmt = conn.prepareStatement
         ("SELECT name, region_of(state)"
         "as region, sales FROM sales_emps"
         "WHERE region_of(state) > ? AND"
         "sales IS NOT NULL"
         "ORDER BY sales DESC");
      stmt.setInt(1, regionParm);
      rs[0] = stmt.executeQuery();
      return;
   }
   catch (SQLException e)
      System.err.println("SQLException:"
      + e.getErrorCode() + e.getMessage());
   }
   return;
}
```

orderedEmps returns a single result set. You can also write methods that return multiple result sets. For each result set returned, you must:

- Include a separate ResultSet array parameter in the method signature.
- Create a Statement object for each result set.
- Assign each result set to the first element of its ResultSet array.

Adaptive Server always returns the current open ResultSet object for each Statement object. When creating Java methods that return result sets:

- Create a Statement object for each result set that is to be returned to the client.
- Do not explicitly close ResultSet and Statement objects. Adaptive Server closes them automatically.

  Note Adaptive Server ensures that ResultSet and Statement objects are not closed by garbage collection unless and until the affected result sets have been processed and returned to the client.

- If some rows of the result set are fetched by calls of the Java next() method, only the remaining rows of the result set are returned to the client.

Creating the SQLJ stored procedure

When you create a SQLJ stored procedure that returns result sets, you must specify the maximum number of result sets that can be returned. In this example, the ranked_emps procedure returns a single result set.

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps(int,
       ResultSet[]'
```

If ranked_emps generates more result sets than are specified by create procedure, a warning displays and the procedure returns only the number of result sets specified. As written, the ranked_emps SQLJ stored procedures matches only one Java method.

Note Some restrictions apply to method overloading when you infer a method signature involving result sets. See “Mapping Java and SQL datatypes” on page 113 for more information.

Calling the procedure

After you have installed the method's class in the database and created the SQLJ stored procedure that references the method, you can call the procedure. You can write the call using any mechanism that processes SQL result sets.

For example, to call the ranked_emps procedure using JDBC, enter the following:

```
java.sql.CallableStatement stmt =
    conn.prepareCall("{call ranked_emps(?)}");
stmt.setInt(1,3);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    String name = rs.getString(1);
    int.region = rs.getInt(2);
    BigDecimal sales = rs.get.BigDecimal(3);
    System.out.print("Name = " + name);
    System.out.print("Region = "+ region);
    System.out.print("Sales = "+ sales);
    System.out.printIn():
}
```

The ranked_emps procedure supplies only the parameter declared in the create procedure statement. The SQL system supplies an empty array of ResultSet parameters and calls the Java method, which assigns the output result set to the array parameter. When the Java method completes, the SQL system returns the result set in the output array element as a SQL result set.

Note You can return result sets from a temporary table only when using an external JDBC driver such as jConnect. You cannot use the Adaptive Server native JDBC driver for this task.

Deleting a SQLJ stored procedure name

You can delete the SQLJ stored procedure name for a Java method using the drop procedure command. For example, enter:

```
drop procedure correct_states
```

which deletes the correct_states procedure name and its reference to the SQLJExamples.correctStates method. drop procedure does not affect the Java class and method referenced by the procedure.

Viewing information about SQLJ functions and procedures

Several system stored procedures can provide information about SQLJ routines:

- sp_depends lists database objects referenced by the SQLJ routine and database objects that reference the SQLJ routine.
- sp_help lists each parameter name, type, length, precision, scale, parameter order, parameter mode and return type of the SQLJ routine.
- sp_helpjava lists information about Java classes and JARs installed in the database. The depends parameter lists dependencies of specified classes that are named in the external name clause of the SQLJ create function or SQLJ create procedure statement.
- sp_helprotect reports the permissions of SQLJ stored procedures and SQLJ functions.

See the *Reference Manual: Procedures* for complete syntax and usage information for these system procedures.

Advanced topics

The following topics present a detailed description of SQLJ topics for advanced users.

Mapping Java and SQL datatypes

When you create a stored procedure or function that references a Java method, the datatypes of input and output parameters or result sets must not conflict when values are converted from the SQL environment to the Java environment and back again. The rules for how this mapping takes place are consistent with the JDBC standard implementation. They are shown below and in Table 6-1 on page 114.

Each SQL parameter and its corresponding Java parameter must be mappable. SQL and Java datatypes are mappable in these ways:

- A SQL datatype and a primitive Java datatype are *simply mappable* if so specified in Table 6-1.
- A SQL datatype and a non-primitive Java datatype are *object mappable* if so specified in Table 6-1.
- A SQL abstract datatype (ADT) and a non-primitive Java datatype are *ADT mappable* if both are the same class or interface.
- A SQL datatype and a Java datatype are *output mappable* if the Java datatype is an array and the SQL datatype is simply mappable, object mappable, or ADT mappable to the Java datatype. For example, character and String[ ] are output mappable.
- A Java datatype is *result-set mappable* if it is an array of the result set-oriented class: java.*sql*.ResultSet.

In general, a Java method is mappable to SQL if each of its parameters is mappable to SQL and its result set parameters are result-set mappable and the return type is either mappable (functions) or void or int (procedures).

Support for int return types for SQLJ stored procedures is a Sybase extension of the SQLJ Part 1 standard.

Table 6-1: Simply and object mappable SQL and Java datatypes

	Corresponding Java datatypes	
SQL datatype	**Simply mappable**	**Object mappable**
char/unichar		java.lang.String
nchar		java.lang.String
varchar/univarchar		java.lang.String
nvarchar		java.lang.String
text		java.lang.String
numeric		java.math.BigDecimal
decimal		java.math.BigDecimal
money		java.math.BigDecimal
smallmoney		java.math.BigDecimal
bit	boolean	Boolean
tinyint	byte	Integer
smallint	short	Integer
integer	int	Integer
bigint	long	java.math.BigInteger
unsigned smallint	int	Integer
unsigned int	long	Integer

SQL datatype	Corresponding Java datatypes	
	Simply mappable	Object mappable
unsigned bigint		java.math.BigInteger
real	float	Float
float	double	Double
double precision	double	Double
binary		byte[]
varbinary		byte[]
datetime		java.sql.Timestamp
smalldatetime		java.sql.Timestamp
date		java.sql.Date
time		java.sql.Time

Specifying Java method signatures explicitly or implicitly

When you create a SQLJ function or stored procedure, you typically specify a Java method signature. You can also allow Adaptive Server to infer the Java method signature from the routine's SQL signature according to standard JDBC datatype correspondence rules described earlier in this section and in Table 6-1.

Sybase recommends that you include the Java method signature as this practise ensures that all datatype translations are handled as specified.

You can allow Adaptive Server to infer the method signature for datatypes that are:

- Simply mappable
- ADT mappable
- Output mappable
- Result-set mappable

For example, if you want Adaptive Server to infer the method signature for correct_states, the create procedure statement is:

```
create procedure correct_states(old char(20),
       not_old char(20))
   modifies sql data
   language java parameter style java
   external name 'SQLJExamples.correctStates'
```

Adaptive Server infers a Java method signature of java.lang.String and java.lang.String. If you explicitly add the Java method signature, the create procedure statement looks like this:

```
create procedure correct_states(old char(20),
       not_old char(20))
    modifies sql data
    language java parameter style java
    external name 'SQLJExamples.correctStates
       (java.lang.String, java.lang.String)'
```

You *must* explicitly specify the Java method signature for datatypes that are object mappable. Otherwise, Adaptive Server infers the primitive, simply mappable datatype.

For example, the SQLJExamples.job method contains a parameter of type int. (See "Handling null argument values" on page 100.) When creating a function referencing that method, Adaptive Server infers a Java signature of int, and you need not specify it.

However, suppose the parameter of SQLJExamples.job was Java Integer, which is the object-mappable type. For example:

```
public class SQLJExamples {
    public static String job(Integer jc)
           throws SQLException ...
```

Then, you must specify the Java method signature when you create a function that references it:

```
create function job_of(jc integer)
...
external name
    'SQLJExamples.job(java.lang.Integer)'
```

Returning result sets and method overloading

When you create a SQLJ stored procedure that returns result sets, you specify the maximum number of result sets that can be returned.

If you specify a Java method signature, Adaptive Server looks for the single method that matches the method name and signature. For example:

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
    language java parameter style java
    external name 'SQLJExamples.orderedEmps
       (int, java.sql.ResultSet[])'
```

In this case, Adaptive Server resolves parameter types using normal Java overloading conventions.

Suppose, however, that you do not specify the Java method signature:

```
create procedure ranked_emps(region integer)
    dynamic result sets 1
```

```
language java parameter style java
external name 'SQLJExamples.orderedEmps'
```

If two methods exist, one with a signature of int, RS[], the other with a signature of int, RS[ ], RS[ ], Application Server cannot distinguish between the two methods and the procedure fails. If you allow Adaptive Server to infer the Java method signature when returning result sets, make sure that *only one method* satisfies the inferred conditions.

Note The number of dynamic result sets specified only affects the maximum number of results that can be returned. It does not affect method overloading.

Ensuring signature validity

If an installed class has been modified, Adaptive Server checks to make sure that the method signature is valid when you invoke a SQLJ procedure or function that references that class. If the signature of a modified method is still valid, the execution of the SQLJ routine succeeds.

Using the command main method

In a Java client, you typically begin Java applications by running the Java Virtual Machine (VM) on the command main method of a class. The JDBCExamples class, for example, contains a main method. It is the command main method that executes when you execute the class from the command line as in the following:

```
java JDBCExamples
```

Note You cannot reference a Java main method in a SQLJ create function statement.

If you reference a Java main method in a SQLJ create procedure statement, the command main method must have the Java method signature `String[]` as in:

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

If the Java method signature is specified in the create procedure statement, it must be specified as `(java.lang.String[])`. If the Java method signature is not specified, it is assumed to be `(java.lang.String[])`.

If the SQL procedure signature contains parameters, those parameters must be char, unichar, varchar, or univarchar. At runtime, they are passed as a Java array of java.lang.String.

Each argument you provide to the SQLJ procedure must be char, unichar, varchar, univarchar, or a literal string because it is passed to the main method as an element of the java.lang.String array. You cannot use the dynamic result sets clause when creating a main procedure.

SQLJ and Sybase implementation: a comparison

This section describes differences between SQLJ Part 1 standard specifications and the Sybase proprietary implementation for SQLJ stored procedures and functions.

Table 6-2 describes Adaptive Server enhancements to the SQLJ implementation.

Table 6-2: Sybase enhancements

Category	SQLJ standard	Sybase implementation
create procedure command	Supports only Java methods that do not return values. The methods must have void return type.	Supports Java methods that allow an integer value return. The methods referenced in create procedure can have either void or integer return types.
create procedure and create function commands	Supports only SQL datatypes in create procedure or create function parameter list.	Supports SQL datatypes and nonprimitive Java datatypes as abstract data types (ADTs).
SQLJ function and SQLJ procedure invocation	Does not support implicit SQL conversion to SQLJ datatypes.	Supports implicit SQL conversion to SQLJ datatypes.
SQLJ functions	Does not allow SQLJ functions to run on remote servers.	Allows SQLJ functions to run on remote servers using Sybase OmniConnect capabilities.
drop procedure and drop function commands	Requires complete command name: drop procedure or drop function.	Supports complete function name and abridged names: drop proc and drop func.

Table 6-3 describes SQLJ standard features not included in the Sybase implementation.

Table 6-3: SQLJ features not supported

SQLJ category	SQLJ standard	Sybase implementation
create function command	Allows users to specify the same SQL name for multiple SQLJ functions.	Requires unique names for all stored procedure and functions.
utilities	Supports sqlj.install_jar, sqlj.replace_jar, sqlj.remove_jar, and similar utilities to install, replace, and remove JAR files.	Supports the installjava utility and the remove java Transact-SQL command to perform similar functions.

Table 6-4 describes the SQLJ standard features supported in part by the Sybase implementation.

Table 6-4: SQLJ features partially supported

SQLJ category	SQLJ standard	Sybase implementation
create procedure and create function commands	Allows users to install different classes with the same name in the same database if they are in different JAR files.	Requires unique class names in the same database.
create procedure and create function commands	Supports the key words no *sql*, contains *sql*, reads *sql* data, and modifies *sql* data to specify the SQL operations the Java method can perform.	Supports modifies *sql* data only.
create procedure command	Supports java.*sql*.ResultSet and the SQL/OLB iterator declaration.	Supports java.*sql*.ResultSet only.
drop procedure and drop function commands	Supports the key word restrict, which requires the user to drop all SQL objects (tables, views, and routines) that invoke the procedure or function before dropping the procedure or function.	Does not support the restrict key word and functionality.

Table 6-5 describes the SQLJ implementation-defined features in the Sybase implementation.

Table 6-5: SQLJ features defined by the implementation

SQLJ category	SQLJ standard	Sybase implementation
create procedure and create function commands	Supports the deterministic | not deterministic keywords, which specify whether or not the procedure or function always returns the same values for the out and inout parameters and the function result.	Supports only the syntax for deterministic | not deterministic, not the functionality.

SQLJ category	SQLJ standard	Sybase implementation
create procedure and create function commands	The validation of the mapping between the SQL signature and the Java method signature can be performed either when the create command is executed or when the procedure or function is invoked. The implementation defines when the validation is performed.	If the referenced class has been changed, performs all validations when the create command is executed, which enables faster execution.
create procedure and create function commands	Can specify the create procedure or create function commands within deployment descriptor files or as SQL DDL statements. The implementation defines which way (or ways) the commands are supported.	Supports create procedure and create function as SQL DDL statements outside of deployment descriptors.
Invoking SQLJ routines	When a Java method executes a SQL statement, any exception conditions are raised in the Java method as a Java exception of the Exception.*sql*Exception subclass. The effect of the exception condition is defined by the implementation.	Follows the rules for Adaptive Server JDBC.
Invoking SQLJ routines	The implementation defines whether a Java method called using a SQL name executes with the privileges of the user who created the procedure or function or those of the invoker of the procedure or function.	SQLJ procedures and functions inherit the security features of SQL stored procedures and Java-SQL functions, respectively.
drop procedure and drop function commands	Can specify the drop procedure or drop function commands within deployment descriptor files or as SQL DDL statements. The implementation defines which way (or ways) the commands are supported.	Supports create procedure and create function as SQL DDL statements outside of deployment descriptors.

SQLJExamples class

This section displays the SQLJExamples class used to illustrate SQLJ stored procedures and functions.

```
import java.lang.*;
import java.sql.*;
import java.math.*;

static String _url = "jdbc:default:connection";

public class SQLExamples {

    public static int region(String s)
           throws SQLException {
       s = s.trim();
       if (s.equals("MN") || s.equals("VT") ||
           s.equals("NH") ) return 1;
       if (s.equals("FL") || s.equals("GA") ||
           s.equals("AL") ) return 2;
       if (s.equals("CA") || s.equals("AZ") ||
           s.equals("NV") ) return 3;
       else throw new SQLException
           ("Invalid state code", "X2001");

    }
    public static void correctStates
       (String oldSpelling, String newSpelling)
       throws SQLException {

       Connection conn = null;
       PreparedStatement pstmt = null;
       try {
           Class.forName
               ("sybase.asejdbc.ASEDriver");
           conn = DriverManager.getConnection(_url);
       }
       catch (Exception e) {
           System.err.println(e.getMessage() +
               ":error in connection");
       }
       try {
           pstmt = conn.prepareStatement
               ("UPDATE sales_emps SET state = ?
               WHERE state = ?");
           pstmt.setString(1, newSpelling);
           pstmt.setString(2, oldSpelling);
           pstmt.executeUpdate();
       }
       catch (SQLException e) {
           System.err.println("SQLException: "+
           e.getErrorCode() + e.getMessage());
```

```
    }

}
public static String job(int jc)
       throws SQLException {
    if (jc==1) return "Admin";
    else if (jc==2) return "Sales";
    else if (jc==3) return "Clerk";
    else return "unknown jobcode";
}
public static String job(int jc)
       throws SQLException {
if (jc==1) return "Admin";
else if (jc==2) return "Sales";
else if (jc==3) return "Clerk";
else return "unknown jobcode";
}
public static void bestTwoEmps(String[] n1,
    String[] id1, int[] r1,
    BigDecimal[] s1, String[] n2,
    String[] id2, int[] r2, BigDecimal[] s2,
    int regionParm) throws SQLException {

n1[0] = "****";
id1[0] = "";
r1[0] = 0;
s1[0] = new BigDecimal(0):
n2[0] = "****";
id2[0] = "";
r2[0] = 0;
s2[0] = new BigDecimal(0);

try {
    Connection conn = DriverManager.getConnection
        ("jdbc:default:connection");
    java.sql.PreparedStatement stmt =
        conn.prepareStatement("SELECT name, id,"
        + "region_of(state) as region, sales FROM"
        + "sales_emps WHERE"
        + "region_of(state)>? AND"
        + "sales IS NOT NULL ORDER BY sales DESC");
    stmt.setInteger(1, regionParm);
    ResultSet r = stmt.executeQuery();

    if(r.next()) {
        n1[0] = r.getString("name");
```

```
            id1[0] = r.getString("id");
            r1[0] = r.getInt("region");
            s1[0] = r.getBigDecimal("sales");
        }
        else return;

        if(r.next()) {
            n2[0] = r.getString("name");
            id2[0] = r.getString("id");
            r2[0] = r.getInt("region");
            s2[0] = r.getBigDecimal("sales");
        }
        else return;
    }
    catch (SQLException e) {
        System.err.println("SQLException: "+
            e.getErrorCode() + e.getMessage());
        }
    }

    public static void orderedEmps
           (int regionParm, ResultSet[] rs) throws
           SQLException {

        Connection conn = null;
        PreparedStatement pstmt = null;

        try {
            Class.forName
                ("sybase.asejdbc.ASEDriver");
            Connection conn =
                DriverManager.getConnection
                ("jdbc:default:connection");
        }
        catch (Exception e) {
            System.err.println(e.getMessage()
                + ":error in connection");
        }

        try {
            java.sql.PreparedStatement
                stmt = conn.prepareStatement
                ("SELECT name, region_of(state)"
                "as region, sales FROM sales_emps"
                "WHERE region_of(state) > ? AND"
                "sales IS NOT NULL"
```

```
                    "ORDER BY sales DESC");
                stmt.setInt(1, regionParm);
                rs[0] = stmt.executeQuery();
                return;
            }
            catch (SQLException e) {
                System.err.println("SQLException:"
                + e.getErrorCode() + e.getMessage());
            }
            return;
        }  return;
        }
    }
```

CHAPTER 7

Debugging Java in the Database

All PCA /JVMs include built-in support for the Java Platform Debugger Architecture (JPDA). The JPDA lets you debug Java code running on Adaptive Server. The JPDA consists of:

- The user interface controlling the debugging, that is, the debugger
- The JVM running the classes to be debugged, and the debug agent providing access to the JVM
- A communication channel between the debug agent and the debugger

The JPDA allows users to debug Java classes either from the command line, by starting the JVM within a debugger application, or remotely, by attaching a debugger to the debug agent on a running JVM. Because users do not have access to the JVM command line in the server, all debugging for Java in the Adaptive Server database is done remotely.

Supported Java debuggers

Every JDK provides an implementation of the basic, command line debugger “jdb” in its development tools package. You can also use an integrated development environment (IDE) for Java development and debugging, for example, Sun Java Studio, IBM WebSphere Studio, JBuilder, and Eclipse. In addition, there are standalone JPDA debuggers such as JSwat.

If you use an IDE or standalone debugger tool, consult the documentation provided by the vendor for specific JDK requirements.

Note The jdb debugger is not included in the JRE distribution. To use jdb, you must install the JDK, which lets you access the jdb debugger.

Setting up Java debugging

Whether you use an IDE, a standalone debugger, or a jdb debugger, you must:

1 Configure the server to support debugging

2 Attach the remote debugger to the JVM debugging agent

Configuring the server to support debugging

Start the debug agent for the JVM using a user-supplied or default port number. Use sp_jreconfig with these configuration parameters to enable debugging, choose a port number, and specify whether the JVM is immediately suspended:

- pca_jvm_java_dbg_agent_port – enables or disables debugging and establishes the port number on which the debug agent in the JVM listens. If you enable this parameter, the JVM starts with the debug agent running in a manner that allows a remote debugger to attach. By default, the debug agent listens on port 8000. To enable the debug agent and allow debugging using the default port, enter:

```
sp_jreconfig "enable", "pca_jvm_java_dbg_agent_port"
```

  To use a different port, change the port number prior to starting the JVM. Once the JVM is started with the debug agent running, the debug agent listens on that port until the JVM shuts down. To enable debugging and change the port on which the debug agent listens, enter:

```
sp_jreconfig "update", "pca_jvm_java_dbg_agent_port", new_port_number
```

- pca_jvm_java_dbg_agent_suspend – controls whether the JVM suspends on startup when the debug agent is running. By default, pca_jvm_java_dbg_agent_suspend is disabled.

When pca_jvm_java_dbg_agent_suspend is enabled, no Java method can execute until a debugger is attached and the JVM is restarted. Suspending the JVM lets you examine the early initialization of the JVM before any classes are loaded. In general, suspending the JVM is not necessary for debugging user classes.

To enable pca_jvm_java_dbg_agent_suspend, enter:

```
sp_jreconfig "enable", "pca_jvm_java_dbg_agent_suspend"
```

Note Use pca_jvm_java_dbg_agent_suspend with caution. Enabling pca_jvm_java_dbg_agent_suspend causes the JVM to suspend and all Adaptive Server Java tasks to wait until you attach and instruct the JVM to continue via the debugger. Sybase recommends that you start the JVM and run a simple Java command to allow you to attach the debugger rather than enabling pca_jvm_java_dbg_agent_suspend. This allows the JVM to boot, and lets you attach the debugger before executing the class that is to be debugged.

Once the configuration values enabling the debug agent in the JVM are set, the next time the JVM is started the debug agent is available. To disable the debug agent the debug agent, disable the configuration parameters and restart the JVM (the agent cannot be turned off once the JVM has started with the agent running).

Note Do not run the debug agent by default. When the debug agent is running, any debug application with network access to the host can potentially connect with the JVM and gain access to object internal data.

Attaching the remote debugger to the JVM debug agent

A debug session begins when the remote debugger attaches to the debug agent running in Adaptive Server. In addition to the connection information supplied using sp_jreconfig, you must enter the location of the source files for the classes that are to be debugged.

If you are using an IDE or standalone debugger, consult the vendor documentation for instruction on how to attach the remote debugger to the debug agent.

This example assumes you are using a jdb command line debugger. You connect to the debug agent on the machine "myhost" on port 8000 and specify Java source files in the JAR archive *mysource.jar* in your home directory.

```
jdb –attach myhost:8000 –source .:${HOME}/mysource.jar
```

The syntax varies for other debugger tools, but you must always supply connection information and source file locations.

CHAPTER 8

File and Network Access Using Java

This chapter describes and provides examples of file and network access using Java.

Adaptive Server supports both file and network I/O capabilities using java.io, java.net, and java.nio packages.

Note If both file and network I/O are streaming large text documents in and out of the server, you may need to increase the amount of memory available to the JVM. If you are handling large documents, you may need to increase the value of the pci memory size configuration parameter to accommodate larger memory requirements. See "The PCI memory pool" on page 16.

File access using *java.io*

The PCA/JVM supports direct file I/O through the java.io and java.nio packages. These packages allow users to read and write files both to and from the file system.

A clear distinction must be made between the user identity used by the operating system and the user identity used by Adaptive Server.

User identity and permissions

When Adaptive Server starts, the server process executes using the system user ID that started the process. For example, if Adaptive Server is started by a system user ID “sybase”:

```
% ps -Usybase -o user,pid,command

USER       PID   CMD
sybase   20405   /sybase/ASE-15-0/bin/dataserver ...
```

Thus, all interactions between the Adaptive Server process and the operating system are associated with the system user ID that started Adaptive Server.

In the server, however, the situation is different. As each user logs in to the server, the user does so with a user ID defined on the Adaptive Server server. This user ID is distinct from the user ID defined on the host machine—even though it might be expected that a user ID represents the same person on both Adaptive Server and the operating system.

Within the database, users may perform different actions based on the roles assigned to them. It is likely that users logged in to Adaptive Server do not have user accounts on the host machine. Thus, the user account that started the server may be acting as a proxy for any number of database users. For example, suppose two files are to be read by the Adaptive Server users (file permissions are strictly read-only for the user).

```
-r----------1 sybase sybuser    1263 Aug 19 18:54 myfile1.dat
-r----------1 jdoe   sybuser     952 Aug  7  9:02 myfile2.dat
```

If users log in to Adaptive Server to run a Java method that attempts to read these files, the Java file I/O eventually comes down to the functions managed by the host interface:

```
isql -Usa -P...
isql -Ujdoe -P...
isql -Ujanedoe -P...
```

The behavior of the underlying read() runtime function is the same for each user. Every user can read *myfile1.dat*, which is owned by the system user ID “sybase” because the server is identified to the operating system as owned by that user. However, no user can read *myfile2.dat*, even though it appears to be owned by one of the database users, because all database user identities are compressed into a single operating system identity “sybase,” which is associated with the process owner. Thus, file access is denied.

Specifying directories for file I/O: UNIX platforms

You can specify optional, additional permission restrictions on the path using traditional UNIX notation. For example, "u+rw" gives the user read-write access, the group read-only access, and all others are denied access. These restrictions do not affect operating system permissions; a user who allowed read-write access in the configuration statement does not gain write access to a directory that has read-only operating system permissions.

When a mask is not provided, the default mask of 0666 is used for the directory for all write operations including file creation. The mask is not used for read-only operations.

When a mask is provided, a default mask of all zeroes is assumed. This ensures that a mask specified as (u+rw) results in a mask of 0600.

Mask syntax

The work_dir (trusted directory) permission mask:

- Must be placed immediately after the path with no intervening spaces.
- Can define [u]ser, [g]roup, [o]ther, and [a]all masks using the leading character (u, g, o, and a) followed by +, –, =, r, w, and x.

For example:

- (u=rw,go=r) equals 0644
- (ugo+r,u+w) equals 0644
- (ugo+r,u+wx) equals 0755
- (ugo=rwx,go–wx) equals 0755

There are many ways to define masks, but they are always evaluated from left to right. For example, suppose the mask is initially defined as 0777 (ugo=rwx). If you later remove w(rite) and x(ecute) for g(roup) and o(ther), the octal equivalent becomes 0744 and the mask (ugo=rwx,go–wx).

If no mask is specified (when the mask portion is optional), the directory uses the default write mask of 0666.

Valid syntax values are:

u ... user (or owner).
g ... group.
o ... other (or world).
a ... all (sets u, g, and o). For example: (a+rw) turns on read and write for u, g, and o.
+ ... turn on bits.
– ... turn off bits.
= ... replace bits. For example: (u=rw) replaces user.
r ... read bit.
w ... write bit.
x ... execute bit.

Examples

- To add a new working directory path to the pca_jvm_work_dir array, enter:

  ```
  sp_jreconfig "add", "work_dir", "/some/path(u+rw)
  ```

  or,

  ```
  sp_jreconfig "add", "work_dir", "/some/path(u=rw)
  ```

- To delete an existing working directory path from the pca_jvm_work_dir array, enter:

  ```
  sp_jreconfig "delete", "work_dir", "/some/path"
  ```

  When deleting or updating a work_dir array element or path entry, only the path portion is required in the supplied string.

- To modify an existing working directory path in the pca_jvm_work_dir array, enter:

  ```
  sp_jreconfig "update", "work_dir", "/old", "/new"
  ```

- To change the path and update permissions, enter:

```
sp_jreconfig "update", "work_dir", "/some/path(u+rw)", "/some/path(u+w)"
```

- To disable an existing working directory path in the pca_jvm_work_dir array, enter:

  ```
  sp_jreconfig "disable", "work_dir", "/some/path"
  ```

  The last argument is a full or partial string value that identifies an individual work_dir array element, and must be supplied even if there is only one element in the array.

- To clear the entire set of working directory paths in the pca_jvm_work_dir array, enter:

```
sp_jreconfig "array_clear", "work_dir"
```

- To enable the entire array, enter:

```
sp_jreconfig "array_enable", "work_dir"
```

- To disable the entire array, enter:

```
sp_jreconfig "array_disable", "work_dir"
```

Specifying directories for file I/O: Windows platforms

You can specify optional, additional permission restrictions on the path using the following notation.

Mask syntax

In a Windows environment, the following syntax can be added to the end of a working directory definition to define the permission mask:

- /RW – defines read/write permission
- /RO – defines read-only permission
- /NA – defines no access

Examples

- To define *D:\my_work_dir* as trusted with full access, enter:

```
sp_jreconfig "add", "work_dir", "C:\my_work_dir/RW"
```

- To define *D:\my_read_only* as trusted with read-only access, enter:

```
sp_jreconfig "add", "work_dir","D:\my_read_only_dir/RO"
```

- To define *E:\general* as trusted with full access, but disallow access to a subdirectory of *E:\general* called *TOP_SECRET*, enter:

```
sp_jreconfig "add", "work_dir","E:\general/RW;E:\general\TOP_SECRET/NA"
```

Delimit individual directory entries with a semi-colon.

File I/O changes

File I/O in the JVM is controlled primarily through file-open operations. After a file has been opened successfully, additional I/O operations on the file are generally permitted. For security reasons, all file-open requests must be made with an absolute path to the physical file; soft links are not supported. Relative paths are converted to absolute paths before any file I/O operations are attempted. For this reason, it is not possible to set up the *$SYBASE* directory as a soft link. Doing so prevents the JVM from initializing because it cannot open files in *$SYBASE/shared.*

If a file-open operation does not conform to a specific set of rules, the file cannot open. File-open rules are based on:

- Whether or not the file already exists
- Whether or not the file is to be opened for read-only or read-write access
- The location of the file to be opened

Rules for opening existing files

This section describes the rules and checks for opening files on UNIX and Windows platforms.

Note If any check fails, the open file request is denied and an error is reported to the caller.

UNIX platforms

If the user ID associated with the server has permission to access the file, the file can be opened for read-only access if it is in the *$SYBASE/shared* directory. Read access is not allowed for any other $SYBASE directory.

Note Write access, including file creation, is never allowed for any $SYBASE directory.

Files opened for write access are given additional checks before the file open request is granted. Adaptive Server checks that:

- The user issuing the file-open request is the file owner.

- The number of hard links is no more than one. If greater than one, the request fails.
- The file to be opened is in a valid directory location. The request fails if the file is in the $SYBASE directory or not in one of the configured working directories.
- The working directory has been configured with an access mask that allows files to be opened with write access. The default mask is 0666. The mask is not required unless you want a mask other than the default.

Windows platforms

If the user ID associated with the server has permission to access the file, access is granted if:

- The file already exists in the %SYBASE% directory structure, read-only access is allowed, and open-for-write requests receive an ERROR_ACCESS_DENIED error, or
- The file exists or is being created in the Windows %TEMP% directory and read-write access is allowed, or
- The file exists or is being created in a configured work directory (a trusted directory). The access allowed is that defined for the work directory, or
- The file exists or is being created in any subdirectory under a trusted directory. The access allowed is that defined for the parent directory.
- If one trusted directory is nested inside another, then the system examines access to each trusted parent in the target file path and the most restricted access is applied. Thus it is possible to allow read-write access to a trusted directory tree, but then specify read-only or no access for specified directories below it. This behavior is similar to Windows behavior when applying ACLs to files.

Rules for creating files with a file open operation

An open request for a file that does not exist is essentially a file-create operation, and must be handled differently than for a file that already exists. The same location constraints that apply to an existing file being opened for write access apply to a newly created file: if the newly created file is to be in either the *$SYBASE* directory structure or is not contained in a configured working directory, the request fails. In addition, the access mask for the directory must allow the user ID associated with the server process to write to the target directory.

Note Write access, including file creation, is always allowed in the */tmp* directory.

On UNIX platforms – files created with an open request must specify write access and are always opened using the file open flags (O-CREAT | O-EXCL | O-RDWR) and an access mask of (0600). For security reasons, these file open flags and this access mask is always used—without regard to the flags and access mask specified by the file open request. You cannot create files using file open flags that specify the file is to be opened for read-only access. To limit the file size or set disk usage quotas, you must do so at the operating system level.

Final file check

After a file open has passed all file checks and the file is allowed to open, a final check ensures that the opened file matches the file originally requested. This prevents attempts to open files not otherwise allowed that attempt to circumvent the checks. If a file open request fails, an annotation is added to the audit trace and a java.lang.IOException is raised to the calling method. Method-specific handling of the IOException determines whether the exception is visible to the user or handled by an alternate mechanism in the Java code.

File access using *java.net*

Adaptive Server support for java.net and java.nio lets you create client-side Java networking applications in the server. You can create a network Java client application that connects to any server, which effectively enables Adaptive Server to function as a client to external servers.

You can use java.net and java.nio to:

- Download documents from any URL on the Internet.
- Send e-mail messages from inside the server.
- Connect to an external server to save a document and perform file functions such as saving or editing a document.
- Access documents using XML.

Note Use java.net with caution:

- Most objects associated with java.net are not serializable; they cannot be inserted into tables.
- Most I/O-related methods use buffered I/O and are not automatically flushed. These methods, such as PrintWriter, must be flushed explicitly.

Examples

This section provides examples for using socket classes and the URL class. You can:

- Access an external document with XML Query Language (XQL), using the URL class.
- Use the MailTo class to mail a document.

Using socket classes

The Java socket classes allow more sophisticated network transfers than the URL classes. The socket classes let you connect to a specified port on any network host, and use the InputStream and OutputStream classes to read and write the data.

Using the URL classes

You can use the URL classes to:

- Send an e-mail message.
- Download an HTTP document from a Web server. The HTTP document can be a static file or can be dynamically constructed by the Web server.
- Access an external document with XQL.
- Use the mailto:URL class to mail a document.

For example, you can mail a document using the URL class. Your client must be connected to a mail server so that the machine referenced by System Properties (in this example, it is salsa.sybase.com), is running a mail server such as sendmail.

For this example, the steps are:

1 Create a URL object.

2 Set a URLConnection object.

3 Create an OutputStream object from the URL object.

4 Write the mail. For example:

```
import java.io.*;
import java.net.*;
public class MailTo {
   public static void sendIt()
       throws Exception{
       System.getProperty("mail.host", "salsa.sybase.com");
       URL url = new URL("mailto:name@sybase.com");
       URLConnection conn = url.openConnection();
       PrintStream out = new PrintStream(conn.getOutputStream(),true);
       out.println ("From janedoes@sybase.com");
       out.println ("Subject: Works Great!");
       out.println ("Thanks for the example - it works great!");
       out.close();
       System.out.println("Message Sent");
   }
}
```

5 Install mailto:URL for sending e-mail messages from within the database:

```
select MailTo.sendIt()
```

You can also use the URL class to download a document from an HTTP URL. When you start, the client connects to a Web server. The steps are:

1 Create a URL object.

2 Create an InputStream object from the URL object.

3 Use read on the InputStream object to read in the document.

The following code sample reads the entire document into Adaptive Server memory and creates a new InputStream on the document in memory.

```
import java.io.*;
import java.net.*;
public class URLprovess {
          public static InputStream readURL()
              throws Exception {
              URL u = newURL("http://www.xxxx.con");
              InputStream in = u.openStream");
              //This is the same as creating URLConnection, then calling
              //getInputStream(). In Adaptive Server, you must read the entire
              //document into memory, and then create an InputStream on the
              //in-memory copy.
              int n = 0;
              int off = 0;
              byte b() = new byte(50000);
              for)off = 0; (off<b.length512) &&
                  ((n = in.read(b.off,512)! = 1);off+=n) {}
              System.out.prinln("Number of bytes read :" + off);
              in.close();
              ByteArrayInputStream test = new ByteArrayInputStream(b,-,off);
              return (InputStream) test;
          }
}
```

After you create the new InputStream class, you can install this class and use it to read a text file into the database. The following example inserts data into table mytable.

```
create table mytable (cl text)
go
insert into mytable values (URLprocess.readURL())
go
Number of bytes read :40867
select datalength(cl) from mytable
go

----------------------
40867
```

CHAPTER 9

Additional Topics

This chapter presents information on several reference topics.

JDK requirement for Java classes in the server

Java classes that you install and use in the server must be at or below the version of the JVM plugged in to Adaptive Server through the PCA/JVM. The PCA/JVM supports Java 6 and later.

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 162) 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 by reference to the target.
- If the target is a Java-SQL column, then the source is serialized, and that serialization is deep copied to the target.

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 157. 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 157. 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( ) or similar method of the object is invoked and the result is truncated to varchar, which is transferred to the client.

  Note The number of bytes transferred to the client is dependent on the value of the @@stringsize global variable. The default value is 50 bytes. See "Representing Java instances" on page 43 for more information.

- 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 b 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 value of length determined by the size of the column.
  - 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.

Suggestions for improving performance

This section provides guidelines for improving performance when using Java in Adaptive Server.

Minimize the number of calls from SQL to the JVM

Off-the-shelf JVMs, and thus the PCA/JVM, benefit from advances in JVM capabilities and significant optimizations so that they are considerably faster than the internal JVM in Adaptive Server 15.0.2 and earlier. However, propagating a SQL call into Java can still create a bottleneck that can be even more pronounced with the PCA/JVM.

To take advantage of the speed of the PCA/JVM, minimize the number of calls from SQL to the JVM.

Consider the simple Address class:

```
public class Address implements java.io.Serializable {
    private int state;
    private String street;
    private String zip;

    // ...

    public Address()
    {
       // ...
    }

    public Address(String street, String zip, int state)
    {
       this();
       this.setStreet(street);
       this.setZip(zip);
       this.setState(state);
    }

    // ,,,

    public void setStreet(String street)
    (
       // ..
    }

    public void setZip(String zip)
    {
       // ...
    }
}
```

Because of the overhead associated with calls into the JVM, it is significantly faster to use the three-argument constructor from SQL than the zero-argument constructor followed by the set methods for the data members. Thus, this statement:

```
1> declare @a Address
2> select @a=new Address("123 Elm Street", "12345", 10)
```

is more efficient than:

```
1> declare @a Adress
2> select @a = new Address()
3> select @a >> setStreet("123 Elm Street")
4> select @a >> setZip("12345")
5> select @a >> setState(10)
```

Pushing as much processing as possible into the Java without requiring repeated crossing of the SQL-Java interface reduces overhead and more fully exploits the improved capabilities of the JVM.

Use the *java.lang.Thread* class with care

The PCA/JVM supports the java.lang.Thread class, which allows you to create classes that use multithreaded methods in Adaptive Server. Threads created within a Java method compete with Adaptive Server for CPU and other resources. Large numbers or resource-intensive threads can impact overall server performance.

Determine if you are running within the PCA/JVM

In general, it makes little difference whether a class is running under the PCA/JVM or a standalone JVM. You can use boolean logic to verify whether the class is loaded via the Sybase ContextClassLoader. For example:

```
boolean running_in_ase = false;

running_in_ase =
this.getClass().getClassLoader().getName().equals
("sybase.aseutils.ContextClassLoader");

if (running_in_ase)
{
    //in ASE
    ...
}
else
{
    //in a standalone JVM
    ...
}
```

Avoid SQL loops in a multi-engine environment

In a multi-engine environment, certain Java/SQL commands can negatively affect performance. This typically happens when the same Java method executes multiple times within a SQL loop. To avoid this, write Java/SQL commands so that the method and the loop are executed in the VM context:

1 Write loop in Java.

2 Call method from Java-coded loop.

Controlling access to native methods in the PCA/JVM

The Java language lets you use functionality implemented in non-Java languages through the Java Native Interface (JNI) via native methods. Classes using native methods must explicitly load the native library using either the load(String *filename*) or loadLibrary(String *libname*) method as described in both the java.lang.System and java.langRuntime classes. Because these libraries are not stored as controlled objects in the database, some users may consider them less secure.

To prevent unexpected access to native libraries, the PCA/JVM has introduced a system property sybase.allow.native.lib to control the loading of native libraries.

Many Java properties can be set either on the command line or from within the application via the java.lang.System setProperty(String *key*, String *value*) method. However, this is forbidden by the SecurityManager to prevent users from overriding system policy. By default, users cannot load native libraries. If an attempt is made to load a native library or alter the existing property setting, a SecurityException is raised and the load attempt fails.

For example, if you try to load the java.net.ServerSocket class without setting the sybase.allow.native.lib property, the initializer fails because it requires the Socket library to be loaded. The actual Java stack varies. However, it or the client message displays:

```
java.lang.SecurityException: Cannot load native
libraries from within a user Task!
```

This indicates that a required native library has been unable to load.

To enable loading of native libraries, set this property in the sybpcidb database prior to starting the JVM:

```
1> sp_jreconfig "add","pca_jvm_java_option",
"-Dsybase.allow.native.lib=true"

2> go
```

Once sybase.allow.native.lib is set true, the additional property is passed in to the JVM on the command line at JVM startup. This property cannot be changed while the JVM is running. If you no longer need to load libraries, use sp_jreconfig to delete or disable pca_jvm_java_option.

Unsupported 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 manipulation facilities of java.lang.Thread.

Warning! Take care when using methods that spawn child threads. java.lang.Thread objects started within a Java method are scheduled by runtime rather than the Adaptive Server scheduler. If these threads are processor intensive or if large numbers of threads are spawned, server performance can degrade due to competition for processor time by greedy user threads.

Because the PCA/JVM uses a standard Java plug-in, the full class distribution is available to you. In general, methods are supported unless their use risks interference with the operation of the server or other Java tasks.

Java in Adaptive Server does not support the native methods invoked through the Java Native Interface (JNI).

This section lists:

- Unsupported Java methods
- Unsupported java.sql methods

Restricted Java packages, classes, and methods

- Because the JVM runs in headless mode, Java methods requiring user input or output are disabled
- Operations that could interfere with the operations of the server or other JVM tasks are not permitted
- These java.lang.Thread methods are not permitted:
  - interrupt()
  - setPriority ()
  - setName()
  - enumerate()
  - setDaemon()
  - checkAccess()
  - getContextClassLoader()
  - setDefaultExceptionHandler()
  - setContextClassLoader()
  - getStackTrace()
  - getAllStarkTraces()
  - setDefaultUncaughtExceptionHandler()
  - stop()
  - destroy()
  - suspend()
  - resume()
  - Deprecated methods are allowed, but may be unsafe
    - countStackFrames()
- These java.lang.ThreadGroup methods are not permitted:
  - getParent()
  - setDaemon()
  - setMaxPriority()
  - checkAccess()

- enumerate()
- interrupt()
- stop()
- destroy()
- suspend()
- resume()
- Deprecated methods are allowed, but may be unsafe
  - allowThreadSuspension()

- Security issues:
  - You can not override the existing SecurityManager or instantiate other class loaders.
  - The exit() methods in java.lang.System and java.lang.Runtime are not permitted.

Unsupported *java.sql* methods and interfaces

For the Java 6 class distribution, the java.sql package conforms with the JDBC 4.x specification. However, the underlying Sybase implementation is at the JDBC 2.0 level. All JDBC methods included since the JDBC 2.0 specification are not supported. In addition, the following methods specified in JDBC 2.0 are not supported.

- Connection.commit( )
- Connection.getMetaData( )
- Connection.nativeSQL( )
- Connection.rollback( )
- Connection.setAutoCommit( )
- Connection.setCatalog( )
- Connection.setReadOnly( )
- Connection.setTransactionIsolation( )
- DatabaseMetaData.* – DatabaseMetaData is supported except for these methods:

    - deletesAreDetected()
    - getUDTs()
    - insertsAreDetected()
    - updatesAreDetected()
    - othersDeletesAreVisible()
    - othersInsertsAreVisible()
    - othersUpdatesAreVisible()
    - ownDeletesAreVisible()
    - ownInsertsAreVisible()
    - ownUpdatesAreVisible()
- 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.setQueryTimeoutt()

Invoking SQL from Java

Adaptive Server supplies a native JDBC driver, java.sql, that implements JDBC 1.1 and 1.2 specifications, and is compliant with version 2.0. 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 9-1 lists Transact-SQL commands and whether or not you can use them in Java methods. You can find further information on most of these commands in the Sybase *Reference Manual: Commands*.

Table 9-1: 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.
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 function	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.

Command	Status
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 function	Supported.
drop index	Not supported.
drop procedure	Not supported.
drop role	Not supported.
drop rule	Not supported.
drop table	Supported.
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.

Command	Status
save transaction	Not supported.
set	See Table 9-2 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.
writetext	Not supported.

Table 9-2 lists set command options and whether or not you can use them in Java methods.

Table 9-2: 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

set command option	Status
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
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
stringsize	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 affect. This is done to support common coding practises in stored procedures.

Note (2) set commands pertaining to parallel degree are allowed but have no affect. 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 9-3 shows SQL-Java datatype mapping.

Table 9-3: Mapping SQL datatypes to Java types

SQL type	Java type
char	String
varchar	String
nchar	String
nvarchar	String
unichar	String
univarchar	String
unitext	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
bigint	long
unsigned smallint	int
unsigned int	long
unsigned bigint	java.math.BigInteger
bigint	java.math.BigInteger
real	float
float	double
double precision	double
binary	byte[ ]
varbinary	byte[ ]
image	java.io.InputStream
datetime	java.sql.Timestamp
smalldatetime	java.sql.Timestamp
bigdatetime	java.sql.Timestamp
bigtime	java.sql.Time
date	java.sql.Date
time	java.sql.Time

Note The mapping of unsigned bigint to double is an approximation; it will not

provide exact values. For exact values, convert the unsigned bigint value to a string value when passing it to a Java method.

Table 9-4 shows Java-SQL datatype mapping.

Table 9-4: Mapping Java scalar types to SQL datatypes

Java scalar type	SQL type
boolean	bit
byte	tinyint
short	smallint
int	integer
long	bigint
float	real
double	double

Java-SQL identifiers

Description

Java-SQL identifiers are a subset of Java identifiers that can be referenced in SQL.

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 fewer.
- 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 4, "Expressions, Identifiers, and Wildcard Characters," in the *Reference Manual: Building Blocks*.

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.

java_sql_identifier
See Java-SQL identifiers.

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: Commands*.

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 Java-SQL column, 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 take 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 static 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.
- 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 Transact-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 executed by the Java VM.

class A class is the basic element of Java programs, containing a set of field declarations and methods. A class is the master copy that determines the behavior and attributes of each instance of that class. class definition is the definition of an active data type, that specifies a legal set of values and defines a set of methods that handle the values. See **class instance**.

class method See **static method**.

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 Value of the class data type that contains a value for each field of the class and that accepts all methods of the class.

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.

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**.

installed classes Java classes and methods that have been placed in the Adaptive Server system by the installjava utility.

instance method A invoked method that references a specific instance of a class.

interface A named collection of method declarations. A class can implement an interface if the class defines all methods declared in the interface.

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 datatypes Java classes, either user-defined or from the JavaSoft API, or Java primitive datatypes, such as boolean, byte, short, and int.

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 method signature The Java datatype of each parameter of a Java method.

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 157.

Java-SQL variable A SQL variable whose datatype is a Java-SQL class.

Java Virtual Machine (JVM)

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 9-3 on page 158, 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 9-4 on page 159, 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.

pluggable component adaptor/ JVM

A Sybase component that manages service requests between Adaptive Server and the JVM.

pluggable component interface (PCI)

The Adaptive Server Java framework, which lets you, with the help of the PCA/JVM, use a commercially available JVM with Adaptive Server.

pluggable component interface (PCI) Bridge

An Adaptive Server component, and part of the PCI, that enables interaction between the JVM plug-in and Adaptive Server.

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 — See **associated JAR**.

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**.

SQL function signature — The SQL datatype of each parameter of a SQLJ function.

SQL-Java datatype mapping — Conversions between Java and SQL datatypes. See "Datatype mapping between Java and SQL" on page 157.

SQL procedure signature — The SQL datatype of each parameter of a SQLJ procedure.

static method — A method invoked without referencing an object. Static methods affect the whole class, not an instance of the class. Also called a class method.

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.

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**.

Index

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z