



**ADMINISTRATION GUIDE | PUBLIC**

SAP Adaptive Server Enterprise 16.0 SP03

Document Version: 1.0 – 2019-06-06

# Java in SAP Adaptive Server Enterprise

# Content

- 1 An Introduction to Java in the Database. . . . . 6**
- 1.1 Invoke Java Methods in the Database. . . . . 6
- 1.2 Store Java Classes as Datatypes. . . . . 7
- 1.3 Store and Query XML in the Database. . . . . 7
- 1.4 Java Components. . . . . 8
- 1.5 Functional Changes . . . . . 8
  - Changes in Class Distribution. . . . . 8
  - The PCA/JVM Runs in Headless Mode. . . . . 9
  - Changes in Memory Management. . . . . 9
  - Changes in ClassLoader Behavior. . . . . 9
- 1.6 Standards. . . . . 10
- 1.7 Java in the Database: Questions and Answers. . . . . 10
- 2 Manage the Java Environment. . . . . 15**
- 2.1 Components of the Java Environment. . . . . 15
  - The JVM Pluggable Component. . . . . 15
  - Pluggable Component Adapter JVM (PCA/JVM). . . . . 16
  - Pluggable Component Interface (PCI) and the PCI Bridge. . . . . 17
  - The PCI Memory Pool. . . . . 18
  - The sybpcidb Database. . . . . 19
  - How Configuration Values are Organized in sybpcidb. . . . . 19
- 2.2 Configuration Values. . . . . 20
- 2.3 Change Configuration Values in a Running Server. . . . . 22
  - Changing Configuration Values by Restarting the Server. . . . . 22
  - Changing Configuration Values by Before the JVM is Initialized. . . . . 23
  - Changing Configuration Values after the JVM is Initialized. . . . . 23
- 2.4 Restore Default Configuration Values to sybpcidb. . . . . 24
- 2.5 Using Monitor Tables to Display Information About the PCI Bridge. . . . . 25
- 2.6 Transferring Java-SQL Objects to Clients. . . . . 25
- 3 Preparing for and Maintaining Java in the Database. . . . . 27**
- 3.1 Java Classes in the Database. . . . . 27
- 3.2 JDBC Drivers. . . . . 27
- 3.3 SAP JVM Support. . . . . 28
- 3.4 Enable Java . . . . . 28
- 3.5 Install Java Classes in the Database. . . . . 29
  - Using installjava. . . . . 29

	Reference Other Java-SQL Classes. . . . .	31
3.6	View Information About Installed Classes and JARs. . . . .	32
3.7	Download Installed Classes and JARs. . . . .	32
3.8	Remove Classes and JARs. . . . .	32
3.9	Retain Classes. . . . .	33
<b>4</b>	<b>Using Java Classes in SQL. . . . .</b>	<b>34</b>
4.1	Java-SQL Names. . . . .	34
4.2	Using Java Classes as Datatypes. . . . .	35
	Create and Alter Tables with Java-SQL Columns. . . . .	36
	Select, Insert, Update, and Delete Java Objects. . . . .	37
4.3	Invoking SQL from Java. . . . .	38
4.4	Invoking Java Methods in SQL. . . . .	39
	Sample Methods. . . . .	40
	Exceptions in Java-SQL Methods. . . . .	40
	Transact-SQL Commands from Java Methods. . . . .	41
4.5	Represent Java Instances. . . . .	46
4.6	Assignment Properties of Java-SQL Data Items. . . . .	47
	Assignments. . . . .	48
4.7	Datatype Mapping Between Java and SQL Fields. . . . .	50
	Datatype Mapping Between Java and SQL. . . . .	51
4.8	Character Sets for Data and Identifiers. . . . .	53
4.9	Subtypes in Java-SQL Data. . . . .	53
	Widening Conversions. . . . .	53
	Narrowing Conversions. . . . .	54
	Runtime Versus Compile-Time Datatypes. . . . .	54
4.10	References to Fields and Methods of Null Instances. . . . .	55
4.11	Null Values as Arguments to Java-SQL Methods. . . . .	56
4.12	Null Values When Using the SQL convert Function. . . . .	57
4.13	Allowed Conversions. . . . .	57
4.14	Java-SQL String Data. . . . .	58
	Zero-Length Strings. . . . .	58
4.15	Type and Void Methods. . . . .	59
	Java Void Instance Methods. . . . .	59
	Java Void Static Methods. . . . .	60
4.16	Equality and Ordering Operations. . . . .	61
4.17	Evaluation Order and Java Method Calls. . . . .	61
	Columns. . . . .	62
	Variables and Parameters. . . . .	62
	Deterministic Java Functions in Expressions. . . . .	63
4.18	Static Variables in Java-SQL Classes. . . . .	64
4.19	Java Classes in Multiple Databases. . . . .	65

	Cross-Database References. . . . .	65
	Inter-Class Transfers. . . . .	66
	Passing Inter-Class Arguments. . . . .	67
	Temporary and Work Databases. . . . .	67
4.20	Java Classes. . . . .	67
<b>5</b>	<b>Data Access Using JDBC. . . . .</b>	<b>71</b>
5.1	JDBC Concepts and Terminology. . . . .	71
5.2	Differences Between Client- and Server-Side JDBC. . . . .	72
5.3	Permissions. . . . .	72
5.4	Using JDBC to Access Data. . . . .	72
	The main( ) and serverMain( ) Methods. . . . .	73
	Using connector() to Obtain a JDBC Connection. . . . .	75
	Using doAction() to Route the Action to Other Methods. . . . .	75
	Using doSQL() to Execute Imperative SQL Operations. . . . .	75
	Using updater() to Execute an update Statement. . . . .	75
	Using selector() to Execute a select Statement. . . . .	76
	Using caller() to Call a SQL Stored Procedure. . . . .	77
5.5	Error Handling in the Native JDBC Driver. . . . .	78
5.6	The JDBC Examples Class. . . . .	79
<b>6</b>	<b>SQLJ Functions and Stored Procedures. . . . .</b>	<b>84</b>
6.1	Creating a SQLJ Stored Procedure or Function. . . . .	84
6.2	Compliance with SQLJ Part 1 Specifications. . . . .	85
6.3	Security and Permissions. . . . .	85
6.4	SQLJ Examples. . . . .	86
6.5	Invoke Java Methods in SAP ASE. . . . .	87
	Invoke Java Methods Directly with their Java Names. . . . .	87
	Invoke Java Methods Indirectly Using SQLJ. . . . .	87
6.6	SQLJ User-Defined Functions. . . . .	88
	Null Argument Values. . . . .	90
	Delete a SQLJ Function Name. . . . .	92
6.7	SQLJ Stored Procedures. . . . .	92
	Modify SQL Data. . . . .	94
	Input and Output Parameters. . . . .	95
	Result Sets. . . . .	98
6.8	View Information About SQLJ Functions and Procedures. . . . .	101
6.9	Map Java and SQL Datatypes. . . . .	101
	Specifying Java Method Signatures Explicitly or Implicitly. . . . .	104
6.10	The Command main Method. . . . .	105
6.11	SQLJ and SAP Implementation: A Comparison. . . . .	106
6.12	SQLJExamples Class. . . . .	108

<b>7</b>	<b>Debugging Java in the Database.</b>	<b>112</b>
7.1	Setting up Java Debugging.	112
	Configuring the Server to Support Debugging.	112
	Attaching the Remote Debugger to the JVM Debug Agent.	113
<b>8</b>	<b>File and Network Access Using Java.</b>	<b>115</b>
8.1	Accessing Files Using java.io.	115
	User Identity and Permissions.	115
	Specifying Directories for File I/O (UNIX).	116
	Specifying Directories for File I/O (Windows).	118
	File I/O Changes.	118
	Rules for Opening Existing Files.	119
	Rules for Creating Files with a File Open Operation.	120
	Final File Check.	120
8.2	File Access Using java.net.	120
	Examples for Socket Classes and the URL Class.	121
<b>9</b>	<b>Suggestions for Improving Performance.</b>	<b>124</b>
9.1	Minimize the Number of Calls from SQL to the JVM.	124
9.2	Use the java.lang.Thread Class with Care.	125
9.3	Determine if you are Running Within the PCA/JVM.	125
9.4	Avoid SQL Loops in a Multi-Engine Environment.	125
<b>10</b>	<b>Unsupported Java API Packages, Classes, and Methods.</b>	<b>126</b>
10.1	Restricted Java Packages, Classes, and Methods.	126
10.2	Unsupported java.sql Methods and Interfaces.	127
<b>11</b>	<b>Java-SQL Reference Information.</b>	<b>129</b>
11.1	Java-SQL Identifiers.	129
11.2	Java-SQL Class and Package Names.	130
11.3	Java-SQL Column Declarations.	131
11.4	Java-SQL Variable Declarations.	132
11.5	Java-SQL Column References.	133
11.6	Java-SQL Member References.	134
11.7	Java-SQL Method Calls.	135
<b>12</b>	<b>Glossary.</b>	<b>137</b>

# 1 An Introduction to Java in the Database

SAP ASE 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 SAP ASE 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.

Java in SAP ASE allows you to invoke Java methods in the database, store Java classes as datatypes, and store and query XML in the database.

## 1.1 Invoke Java Methods in the Database

You can install Java classes in SAP ASE then invoke the static methods of those classes directly in SQL, or 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, for example, in Transact-SQL `select` lists and `where` clauses.

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

These 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 that is 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.

### Related Information

[Standards \[page 10\]](#)

## 1.2 Store 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 SAP ASE 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.

## 1.3 Store and Query 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* describes the SAP ASE native XML processor and the SAP ASE Java-based XML support, introduces XML in the database, and documents the query and mapping functions that comprise XML Services.

## 1.4 Java Components

SAP ASE lets you plug in commercial, off-the-shelf Java runtime environment (JRE) and Java virtual machine (JVM) components.

After configuring your 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 SAP ASE versions prior to 15.0.3 as well as applications created using the SAP ASE version 15.0.3 and later.

The Java interface for SAP ASE include the commercial JVM and the SAP ASE components that support it:

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

### Related Information

[Manage the Java Environment \[page 15\]](#)

## 1.5 Functional Changes

With SAP ASE version 15.0.3, SAP ASE introduces support for commercial JVMs such as the Sun Java 2 Platform, Standard Edition (J2SE).

SAP ASE version 15.0.2 and earlier provided an internal JVM.

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

### 1.5.1 Changes in Class Distribution

The Java runtime classes delivered with SAP ASE 15.0.2 and earlier was a limited subset of the Java 1.2 release.

SAP ASE 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.

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



SAP ASE 15.0.3 replaces the `runtime.zip` file with the `sybasert.jar` (which contains the SAP ASE 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.

## 1.5.2 The PCA/JVM Runs in Headless Mode

Classes and methods requiring user interaction were excluded from the Java distribution provided by SAP ASE 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.

## 1.5.3 Changes in Memory Management

SAP ASE 15.0.3 and later uses a single PCI memory pool. Any existing configuration values from 15.0.2 and earlier are ignored by SAP ASE 15.0.3. You must specify the total memory for the PCI subsystem using the `pci_memory_size` configuration parameter.

SAP ASE 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.

If you are transitioning from SAP ASE 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 SAP ASE internal JVM. Once the size of the PCI memory pool is appropriately configured, you should see no difference in behavior.

## Related Information

[Manage the Java Environment \[page 15\]](#)

## 1.5.4 Changes in ClassLoader Behavior

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

In SAP ASE 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 SAP ASE 15.0.2 and earlier may not load in SAP ASE 15.0.3 and later unless a full set of user and Java-supplied classes is provided so that all dependencies can be satisfied.

## 1.6 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."

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

## 1.7 Java in the Database: Questions and Answers

SAP ASE is not only extending the capabilities of the database with Java, but also extending the capabilities of Java with the database.

These points are explained in detail:

- 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 SAP ASE databases and, conversely, store XML documents and data extracted from them in SAP ASE 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.

SAP ASE provides a runtime environment for Java classes. You need a Java development environment, such as PowerJ or Sun Microsystems Java Development Kit (JDK), to write and compile Java.

## How is Java Executed in the Database?

When SAP ASE 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 SAP ASE Java framework has been designed to work with any standard JVM that supports Java 6 or later.

SAP ASE 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 SAP ASE 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.

SAP ASE 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 are Java and SQL 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 – SAP ASE 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?

SAP ASE supports all standard Java classes in the database. Because Java in the database runs in headless mode certain methods expecting user input or output devices raise a Java exception.

SAP ASE does not support non-class resources in JAR files within Java in the database.

## 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, SAP ASE 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.

SAP ASE 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 are Java Classes Used 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 SAP ASE database.

### **i** 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 to:
  - 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 Cannot be Done With Java in the Database?

SAP ASE 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.

You should 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.

## 2 Manage the Java Environment

You can plug in off-the-shelf, standard Java JRE and JVM components such as J2SE, to SAP ASE.

The SAP ASE 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 SAP ASE. You can change or upgrade your Java applications to take advantage of new Java functionality as it becomes available.

### 2.1 Components of the Java Environment

The components that make up the SAP ASE Java environment are the JVM plug-in, PCA/JVM, PCI, and the PCI memory pool.



#### 2.1.1 The JVM Pluggable Component

The JVM plug-in is a dynamically loaded module that is engineered, supported, and installed on your platform independently from SAP ASE.

To SAP ASE, the plug-in is a “black-box” application and not SAP ASE-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 SAP ASE.

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

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.

### i 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 SAP ASE 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, SAP ASE 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.

## 2.1.2 Pluggable Component Adapter JVM (PCA/JVM)

The PCA/JVM acts as a broker, managing service requests between the SAP ASE server and the JVM.

The PCA/JVM forwards and controls requests in both directions—from the server to the JVM, and from the JVM to the server.

### i Note

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

### 2.1.2.1 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.lang.Runtime` 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`.

## 2.1.3 Pluggable Component Interface (PCI) and the PCI Bridge

The PCI is a generic interface internal to SAP ASE; it is installed by default when you install or upgrade SAP ASE.

The PCI Bridge, a component internal to the PCI, performs the actual work between SAP ASE 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 SAP ASE error log

For most scenarios, the default PCI Bridge configuration is appropriate and sufficient. If necessary, and with the advice of SAP Product 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`.

## Related Information

[Change Configuration Values in a Running Server \[page 22\]](#)

### 2.1.4 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 SAP ASE 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 the server for the change to take effect.

If the 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 SAP ASE

In a multi-engine environment, multiple SAP ASE 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 SAP ASE 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 SAP ASE configure heap sizes for you, the calculated heap size must be greater than 4MB and you must not set the `-Xmx` and `-Xms` arguments. (SAP ASE uses the values stored in `sybpcidb`.)

When SAP ASE 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.

## 2.1.5 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.

The `sybpcidb` system stored procedures are:

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

The `sybpcidb` database contains these tables:

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

See:

- *Reference Manual: Tables* for more information about `sybpcidb`.
- *Reference Manual: Procedures* for more information about `sp_pciconfig` and `sp_jreconfig`.

## 2.1.6 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.

## 2.2 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 SAP Product 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.

Option	Description
<code>enable pci</code>	Enables the PCI Bridge.
<code>enable java</code>	Enables Java in the database.
<code>pci memory size</code>	Sets the maximum size of the PCI memory pool.

#### i 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 SAP Product Support.

## Configuration Options for the PCA/JVM

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

Argument	Description
<code>pca_jvm_module_path</code>	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.
<code>pca_jvm_work_dir</code>	Add one entry to this argument array for each working directory (trusted directory) that can be configured with a specific permission mask, as needed.
<code>pca_jvm_netio</code>	Enable this argument to enable network I/O. Disable this argument to disable network I/O.
<code>pca_jvm_dbg_agent_port</code>	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.
<code>pca_jvm_java_dbg_agent_suspend</code>	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 <i>Reference Manual: Commands</i> .

### i Note

Use `pca_jvm_java_dbg_agent_suspend` with caution. Enabling `pca_jvm_java_dbg_agent_suspend` suspends the JVM and all SAP ASE Java tasks wait until you attach the debugger and instruct the JVM to continue. SAP ASE 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 SAP Product Support.

## 2.3 Change Configuration Values in a Running Server

If, with advice from SAP Product Support, you want to change the default configuration values, you can use the `sp_jreconfig` and `sp_pciconfig` system stored procedures.

When SAP ASE 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 SAP ASE
- 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)

### Related Information

[Configuration Values \[page 20\]](#)

### 2.3.1 Changing Configuration Values by Restarting the Server

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

### Context

#### i Note

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

### Procedure

1. Use `sp_jreconfig` or `sp_pciconfig` to change configuration values.
2. Restart SAP ASE.

## 2.3.2 Changing Configuration Values by Before the JVM is Initialized

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

### Procedure

1. Use `sp_jreconfig` to change configuration values.
2. Load the configuration parameters into memory. Enter:

```
sp_jreconfig "reload_config"
```

### Results

You do not need to restart SAP ASE for the new configuration values to take effect.

#### i 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 SAP ASE.

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.

## 2.3.3 Changing Configuration Values after the JVM is Initialized

If SAP ASE 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.

### Context

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

## Procedure

1. For threaded or process mode, use `sp_jreconfig` to change configuration values.
2. If SAP ASE is running in multiple-engine process mode and the JVM is running on its own engine process: Bring the engine running the JVM offline, then back online (in this example, engine number three):

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

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

SAP ASE 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.

3. For threaded mode, restart SAP ASE.

## 2.4 Restore 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 are using a single- or multiple-engine SAP ASE server.

### Procedure

1. For single engine or multiple-engine server, reinstall the `installpcidb` installation script to reset the `sybpcidb` configuration table values to their factory defaults. For example:

```
isql -Usa -P<sa_password> -S<server_name>  
-i $SYBASE_ASE/scripts/installpcidb
```

2. For a multiple-engine server, bring the engine running the JVM offline, then back online. For example:

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

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

You do not need to restart SAP ASE for the new configuration values to take effect.

3. For single engine, restart SAP ASE. The default configuration values take effect when the JVM initializes in response to the first Java request.



## 2.5 Using Monitor Tables to Display Information About the PCI Bridge

You can display information about the PCI Bridge using the `monPCIBridge` and `monPCISlots` monitor tables.

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

## 2.6 Transferring Java-SQL Objects to Clients

When a value, which has a datatype that is a Java-SQL object type, is transferred from SAP ASE 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.

- If the client is a Java client that uses `jdbc` 4.0 or later, the server transmits the object serialization to the client. This serialization is seamlessly deserialized by `jdbc` to yield a copy of the object.

- If the client is a `blob` 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.

## Related Information

[Represent Java Instances \[page 46\]](#)

# 3 Preparing for and Maintaining Java in the Database

The SAP ASE runtime environment for Java requires a third-party JVM, the SAP PCI, which is available as part of the database server, and the SAP ASE runtime Java classes, or Java API. If you are running Java applications on the client, you may also require the JDBC driver, jConnect, on the client.

## 3.1 Java Classes in the Database

You can use The standard Java distribution found in `rt.jar` and classes installed in the `ext` directory under the Java installation directory, or user-defined Java classes.

### i 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.

Classes	Description
Runtime Java Classes	They are downloaded automatically when SAP ASE is installed and are available thereafter from <code>\$(SYBASE)/\$(SYBASE_ASE)/lib/sybasert.jar</code> in UNIX ( <code>%SYBASE%\%SYBASE_ASE%\lib\sybasert.jar</code> in Windows). SAP ASE sets the CLASSPATH environment when the JVM starts.

### i Note

If CLASSPATH is set in the operating system environment, SAP ASE ignores that value when the internal JVM starts.

User-Defined Java Classes	After you install user-defined classes into the database using the <code>installjava</code> utility, these classes are available from other classes in the database and from SQL as user-defined datatypes.
---------------------------	---

## 3.2 JDBC Drivers

The SAP native JDBC driver that comes with SAP ASE supports JDBC versions 1.1 and 1.2, and is compliant with several classes and methods of JDBC version 2.0.

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

## Related Information

[Java-SQL Reference Information \[page 129\]](#)

### 3.3 SAP JVM Support

SAP JRE is used to support Java applications.

By default, the SAP JRE is installed in:

```
$SYBASE/shared/SAPJRE-7_*
```

The installer automatically sets the `<SAP_JRE7>`, `<SAP_JRE7_32>`, and `<SAP_JRE7_64>` environment variables.

#### i Note

(IBM AIX only) You must set the data size resource limit to `unlimited` when using any Java application:

```
limit datasize unlimited
```

See your operating system documentation.

### 3.4 Enable Java

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

#### Prerequisites

- Configure `sybpcidb` as described in the installation guide for your platform.
- Enable the PCI before enabling Java.

#### Procedure

1. Enable the server and its databases for Java by using these configuration parameters in `isql`:

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

2. Shut down and restart the server.

## 3.5 Install 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 *Utility Guide* for detailed information about these utilities. Both utilities perform the same tasks; for simplicity, this document uses UNIX examples.

### 3.5.1 Using installjava

`installjava` copies an uncompressed JAR file into the SAP ASE 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*.

#### **i** 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.

### 3.5.1.1 Install 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
```

### 3.5.1.2 Retain 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 SAP ASE 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, SAP ASE 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, SAP ASE does not retain any association of the classes with the JAR. This is the default option.

SAP 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, *Download Installed Classes and JARs*.

## Related Information

[Download Installed Classes and JARs \[page 32\]](#)

### 3.5.1.3 Update 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.
- `update` – you can install a class with the same name as an existing class, and the newly installed class replaces the existing class.

### ⚠ Caution

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.

To reorganize your installed classes in new JARs, it may be easier to first disassociate the affected classes from their JARs.

## Related Information

[Retain Classes \[page 33\]](#)

## 3.5.2 Reference 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.

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.

## Related Information

[Exceptions in Java-SQL Methods \[page 40\]](#)

## 3.6 View Information About Installed Classes and JARs

Use `sp_helpjava`, to view information about classes and JARs installed in the database.

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.

## 3.7 Download 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.

## 3.8 Remove 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"
```



## i Note

SAP ASE 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`.

## 3.9 Retain 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.

# 4 Using Java Classes in SQL

Before you use Java in your SAP ASE database, there are some general considerations.

Java classes contain:

- Fields that have declared Java datatypes.
- Methods for which parameters and results have declared Java datatypes.
- Java datatypes for which there are corresponding SQL datatypes.

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.

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

## Related Information

[Datatype Mapping Between Java and SQL \[page 51\]](#)

### 4.1 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.

## Related Information

[Java-SQL Identifiers \[page 129\]](#)

## 4.2 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  
)
```

## 4.2.1 Create and Alter 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 SAP ASE 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 16 KB, 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 with a runtime representation that is more than the 16KB 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. SAP ASE 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 16KB 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 ] ]...
```

### i Note

You cannot change the column size of an in-row column using `alter column` in this SAP ASE release.

If a table containing Java columns is partitioned, you cannot alter the table without first dropping the partitions. To change the table schema, remove the partitions, use `alter table` command, then repartition the table.

## 4.2.2 Select, Insert, Update, and Delete 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_addr>>street, @zip = home_addr>>zip from emps  
    where name = 'Bob Smith'  
select @name = name from emps  
    where home_addr>>street= '456 Shoreline Drive'  
  
update emps  
    set home_addr>>street = '457 Shoreline Drive',  
        home_addr>>zip = '99323'  
    where home_addr>>street = '456 Shoreline Drive'
```

## 4.3 Invoking SQL from Java

SAP ASE 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 SAP ASE to perform SQL operations.

`java.sql.DriverManager.getConnection( )` accepts the URLs: null, the null string (""), and the `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.

## 4.4 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 SAP ASE 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.

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.

### Related Information

[Security and Permissions \[page 85\]](#)

[Invoke Java Methods in SAP ASE \[page 87\]](#)

## 4.4.1 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%'
```

## Related Information

[Type and Void Methods \[page 59\]](#)

## 4.4.2 Exceptions in Java-SQL Methods

When the invocation of a Java-SQL method completes with unhandled exceptions, a SQL exception is raised.

This error message is displayed:

```
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.



## 4.4.3 Transact-SQL Commands from Java Methods

You can use certain Transact-SQL commands in Java methods called within the SQL system.

### Support Status of Transact-SQL Commands

Command	Status
<code>alter database</code>	Not supported.
<code>alter role</code>	Not supported.
<code>alter table</code>	Supported.
<code>begin ... end</code>	Supported.
<code>begin transaction</code>	Not supported.
<code>break</code>	Supported.
<code>case</code>	Supported.
<code>checkpoint</code>	Not supported.
<code>commit</code>	Not supported.
<code>compute</code>	Not supported.
<code>connect - disconnect</code>	Not supported.
<code>continue</code>	Supported.
<code>create database</code>	Not supported.
<code>create default</code>	Not supported.
<code>create existing table</code>	Not supported.
<code>create function</code>	Supported.
<code>create index</code>	Not supported.
<code>create procedure</code>	Not supported.
<code>create role</code>	Not supported.
<code>create rule</code>	Not supported.
<code>create schema</code>	Not supported.

Command	Status
<code>create table</code>	Supported.
<code>create trigger</code>	Not supported.
<code>create view</code>	Not supported.
<code>cursors</code>	Not supported.
<code>dbcc</code>	Not supported.
<code>declare</code>	Supported.
<code>disk init</code>	Not supported.
<code>disk mirror</code>	Not supported.
<code>disk refit</code>	Not supported.
<code>disk reinit</code>	Not supported.
<code>disk remirror</code>	Not supported.
<code>disk unmirror</code>	Not supported.
<code>drop database</code>	Not supported.
<code>drop default</code>	Not supported.
<code>drop function</code>	Supported.
<code>drop index</code>	Not supported.
<code>drop procedure</code>	Not supported.
<code>drop role</code>	Not supported.
<code>drop rule</code>	Not supported.
<code>drop table</code>	Supported.
<code>drop trigger</code>	Not supported.
<code>drop view</code>	Not supported.
<code>dump database</code>	Not supported.
<code>dump transaction</code>	Not supported.
<code>execute</code>	Supported.
<code>goto</code>	Supported.

<b>Command</b>	<b>Status</b>
<code>grant</code>	Not supported.
<code>group by and having clauses</code>	Supported.
<code>if...else</code>	Supported.
<code>insert table</code>	Supported.
<code>kill</code>	Not supported.
<code>load database</code>	Not supported.
<code>load transaction</code>	Not supported.
<code>online database</code>	Not supported.
<code>order by Clause</code>	Supported.
<code>prepare transaction</code>	Not supported.
<code>print</code>	Not supported.
<code>raiserror</code>	Supported.
<code>readtext</code>	Not supported.
<code>return</code>	Supported.
<code>revoke</code>	Not supported.
<code>rollback trigger</code>	Not supported.
<code>rollback</code>	Not supported.
<code>save transaction</code>	Not supported.
<code>set</code>	<i>See Support Status of set Command Options.</i>
<code>setuser</code>	Not supported.
<code>shutdown</code>	Not supported.
<code>truncate table</code>	Supported.
<code>union Operator</code>	Supported.
<code>update statistics</code>	Not supported.
<code>update</code>	Supported.
<code>use</code>	Not supported.

Command	Status
<code>waitfor</code>	Supported.
<code>where Clause</code>	Supported.
<code>while</code>	Supported.
<code>writetext</code>	Not supported.

## Support Status of set Command Options

Command Option	Status
<code>ansinull</code>	Supported.
<code>ansi_permissions</code>	Supported.
<code>arithabort</code>	Supported.
<code>arithignore</code>	Supported.
<code>chained</code>	Not supported.  set commands with options <code>chained</code> or <code>transaction isolation level</code> are allowed only if the setting that they specify is already in effect. That is, this kind of <code>set</code> command is allowed if it has no effect. This is done to support common coding practises in stored procedures.
<code>char_convert</code>	Not supported.
<code>cis_rpc_handling</code>	Not supported
<code>close on endtran</code>	Not supported
<code>cursor rows</code>	Not supported
<code>datefirst</code>	Supported
<code>dateformat</code>	Supported
<code>fipsflagger</code>	Not supported
<code>flushmessage</code>	Not supported
<code>forceplan</code>	Supported
<code>identity_insert</code>	Supported
<code>language</code>	Not supported

Command Option	Status
<code>lock</code>	Supported
<code>nocount</code>	Supported
<code>noexec</code>	Not supported
<code>offsets</code>	Not supported
<code>or_strategy</code>	Supported
<code>parallel_degree</code>	Supported.  set commands pertaining to parallel degree are allowed but have no effect. This supports the use of stored procedures that set the parallel degree for other contexts.
<code>parseonly</code>	Not supported
<code>prefetch</code>	Supported
<code>process_limit_action</code>	Supported.  set commands pertaining to parallel degree are allowed but have no effect. This supports the use of stored procedures that set the parallel degree for other contexts.
<code>procid</code>	Not supported
<code>proxy</code>	Not supported
<code>quoted_identifier</code>	Supported
<code>replication</code>	Not supported
<code>role</code>	Not supported
<code>rowcount</code>	Supported
<code>scan_parallel_degree</code>	Supported.  set commands pertaining to parallel degree are allowed but have no effect. This supports the use of stored procedures that set the parallel degree for other contexts.
<code>self_recursion</code>	Supported
<code>session_authorization</code>	Not supported
<code>showplan</code>	Supported
<code>sort_resources</code>	Not supported

Command Option	Status
<code>statistics io</code>	Not supported
<code>statistics subquerycache</code>	Not supported
<code>statistics time</code>	Not supported
<code>string_rtruncation</code>	Supported
<code>stringsize</code>	Supported
<code>table count</code>	Supported
<code>textsize</code>	Not supported
<code>transaction iso level</code>	Not supported.  set commands with options <code>chained</code> or <code>transaction isolation level</code> are allowed only if the setting that they specify is already in effect. That is, this kind of <code>set</code> command is allowed if it has no effect. This is done to support common coding practises in stored procedures.
<code>transactional_rpc</code>	Not supported

## 4.5 Represent 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, SAP ASE must convert the object to a viewable representation.

To use an actual string value, SAP ASE 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.

### i 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, SAP ASE imposes a limit on the number of bytes returned. SAP ASE 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
```

## 4.6 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 is set to `null`, in the same manner as any other column with a value that is not specified in an `insert`. This `null` value is generated entirely in SQL, and initialization of the `mailing_addr` column does not involve the Java VM at all.

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

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

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

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

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

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

- The Java-SQL column `home_addr` contains a serialized representation of the value of an `Address` instance. The SQL system invokes the Java VM to 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'
```

## 4.6.1 Assignments

Rules are defined for assignment between SQL data items with datatypes that 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.

### i 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*) yield a reference to an object in the Java VM. Thus, the source is a reference to an object in the Java VM.

## Related Information

[Java-SQL Column References \[page 133\]](#)

### 4.6.1.1 Assignment Rules at Compile-Time

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.
- `SC_T` must be the same as `TC_T` or a subclass of `TC_T`.

### 4.6.1.2 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.

## 4.7 Datatype Mapping Between Java and SQL Fields

When you transfer data in either direction between the Java VM and SAP ASE, you must take into account that the datatypes of the data items are different in each system.

SAP ASE automatically maps SQL items to Java items and vice versa according to the correspondence tables in *Datatype Mapping Between Java and SQL*.

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 16KB bytes is defined by SAP ASE.
  - 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 with a datatype that 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 necessary, truncated to 255 characters.

### Related Information

[Datatype Mapping Between Java and SQL \[page 51\]](#)

## 4.7.1 Datatype Mapping Between Java and SQL

SAP ASE maps SQL datatypes to Java types (SQL-Java datatype mapping) and Java scalar types to SQL datatypes (Java-SQL datatype mapping).

Table 1: 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

SQL Type	Java Type
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 2: 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

## 4.8 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.

### i 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.

## 4.9 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.

### 4.9.1 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.

## 4.9.2 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 `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
```

## 4.9.3 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.

## Related Information

[Null Values When Using the SQL convert Function \[page 57\]](#)

### 4.10 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, which has a “`home_addr`” column that 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.

## 4.11 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.



## 4.12 Null Values When Using the SQL convert Function

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*, 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.

SAP ASE 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 in which the `home_addr` column is an `Address2Line` and a null for those rows in which the `home_addr` column is an `Address`. The `select` also shows a null `line2` value for those rows in which the `home_addr` column is null.

### Related Information

[Subtypes in Java-SQL Data \[page 53\]](#)

## 4.13 Allowed Conversions

You can use `convert` to change the expression datatype in different 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*. 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*. 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.

## Related Information

[Datatype Mapping Between Java and SQL Fields \[page 50\]](#)

[Datatype Mapping Between Java and SQL \[page 51\]](#)

## 4.14 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 that is a `char`, `varchar`, `nchar`, `nvarchar`, or `text` type, 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 that is a `char`, `varchar`, `nchar`, or `text` type 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.

### 4.14.1 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 that has a length of zero is assigned to a SQL data item that is a `char`, `varchar`, `nchar`, `nvarchar`, or `text` type, the Java-SQL `String` value is replaced with a single space.

For example:

```
declare @s varchar(20)
select @s = new java.lang.String()
select @s, char_length(@s)
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.

## 4.15 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*, where its type is `String`.
- The `removeLeadingBlanks( )` method is a *void method*.
- The `Address` constructor method is a *type method*, where its 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.

### 4.15.1 Java Void Instance Methods

To use the "update-in-place" actions of Java void instance methods in the SQL system, Java in SAP ASE treats a call of a Java void instance method in these ways.

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 returns the `street` field that has the leading blanks removed:

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

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: `removeLeadingBlanks( )` 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.

## 4.15.2 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.

## 4.16 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.

## 4.17 Evaluation Order and Java Method Calls

SAP ASE does not have a defined order for evaluating operands of comparisons and other operations. Instead, SAP ASE 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;}
    }
}
```

## 4.17.1 Columns

In general, avoid invoking in the same SQL statement multiple methods on the same Java-SQL object. If at least one of the statements modifies the object, the order of evaluation can affect the outcome.

In this example, the `where` clause passes the same `home_addr` column in two different method invocations:

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

Consider the evaluation of the `where` clause for a row with a `home_addr` column that has a five-character code, such as "95123."

SAP ASE 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, SAP ASE may let the second invocation see the modifications of the argument made by the first invocation.

In the example, the first invocation chosen by SAP ASE 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.

## 4.17.2 Variables and Parameters

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."

## 4.17.3 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 SAP ASE are deterministic.

As a result, if the parameters and input values in an expression involving a Java function do not change, SAP ASE treats the entire expression as deterministic.

When SAP ASE encounters a Java function in an expression, SAP ASE 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, SAP ASE “precalculates” the result, and populates the array with both values.

You can make expressions nondeterministic by adding a reference to columns so that SAP ASE 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.

## 4.18 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.recommendedLimitselect Address>>recommendedLimit

```

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

SAP ASE uses a separate JVM thread for each SAP ASE task within the same JVM. All user classes are loaded by `ClassLoaders` associated only with the specific SAP ASE task executing the particular Java method. Because `ClassLoaders` associated with user classes are not shared across SAP ASE tasks, user classes are not considered the same. Therefore, class variables from user classes are not visible across SAP ASE tasks.



However, class variables from classes loaded by the system ClassLoader are visible across all SAP ASE 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.

In the Cluster Edition, SAP ASE handles class variables from user classes and classes loaded by the system ClassLoader. 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 SAP ASE task can run across multiple nodes, if user classes rely on class variables, you must explicitly set that same class variable on all nodes.

## 4.19 Java Classes in Multiple Databases

You can store Java classes of the same name in different databases in the same SAP ASE system.

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 SAP ASE 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.

### 4.19.1 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`.

## 4.19.2 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 with a declared type that 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. This example shows a fragment in a stored procedure on `db2`:

```
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 SAP ASE.
- 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.

## 4.19.3 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.

## 4.19.4 Temporary and Work Databases

All rules for Java classes and databases 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.

## 4.20 Java Classes

Simple Java classes are used to illustrate Java in SAP ASE.

This is the `Address` class:

```
//
// Copyright (c) 2015
// SAP SE
// 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(zip);
    }
}

```

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

```

//
// Copyright (c) 2015
// SAP SE
// Dublin, CA 94568
// All Rights Reserved
//
/**
 * A subclass of the Address class that adds a second 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) 2015
// SAP SE
// 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 "";
    }
}

```

# 5 Data Access Using JDBC

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 SAP ASE SQL interface in either of two ways:

- *JDBC on the client* – Java client applications can make JDBC calls to SAP ASE using the SAP 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 SAP ASE.

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.

## 5.1 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 procedure for using JDBC is as follows.

Step	Description
Create a <code>&lt;Connection&gt;</code> object	Call the <code>getConnection( )</code> static method of the <code>DriverManager</code> class to create a <code>&lt;Connection&gt;</code> object. This establishes a database connection.
Generate a <code>&lt;Statement&gt;</code> object	Use the <code>&lt;Connection&gt;</code> object to generate a <code>&lt;Statement&gt;</code> object.
Pass a SQL statement to the <code>&lt;Statement&gt;</code> object	If the statement is a query, this action returns a <code>&lt;ResultSet&gt;</code> object. The <code>&lt;ResultSet&gt;</code> object contains the data returned from the SQL statement, but provides it one row at a time (similar to the way a cursor works).
Loop over the rows of the results set	Call the <code>next( )</code> method of the <code>&lt;ResultSet&gt;</code> object to: <ul style="list-style-type: none"><li>• Advance the current row (the row in the result set that is being exposed through the <code>&lt;ResultSet&gt;</code> object) by one row.</li><li>• Return a Boolean value (true/false) to indicate whether there is a row to advance to.</li></ul>
For each row, retrieve the values for columns in the <code>&lt;ResultSet&gt;</code> object	Use the <code>getInt( )</code> , <code>getString( )</code> , or similar method to identify either the name or position of the column.

## 5.2 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 SAP 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 SAP ASE 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.

## 5.3 Permissions

Java classes are executed with the permissions of the connection executing them, and classes containing JDBC statements can be accessed by any user.

- *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.

## 5.4 Using JDBC to Access Data

You can use JDBC to perform the typical operations of a SQL application. To execute the JDBC 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 SAP ASE.



## i Note

The SAP ASE native driver does not support `create procedure` and `drop procedure` statements; use the `jConnect` client to create or drop stored procedures.

`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`.

`JDBC Example` operates only on the `xmp` table and `inoutproc` procedure.

## 5.4.1 The `main( )` and `serverMain( )` Methods

`JDBC Examples` has two primary methods: `main` and `serverMain`.

- `main( )` – is invoked from the command line of the `jConnect` client.
- `serverMain( )` – performs the same actions as `main( )`, but is invoked within SAP ASE.

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

### 5.4.1.1 Using `main( )`

You can invoke the `main( )` method from a `jConnect` command line.

```
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
```

## 5.4.1.2 Using serverMain( )

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.

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
```

### i Note

Server-side calls of `serverMain( )` do not require a `<server-name:port-number>` parameter; SAP ASE simply connects to itself.

## 5.4.2 Using connector() to Obtain a JDBC Connection

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

Both `main()` and `serverMain()` call `connector()` 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 `connector()` method, `JDBCExamples`' other methods are independent of their server- or client-side environment.

## 5.4.3 Using doAction() to Route the Action to Other Methods

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.

## 5.4.4 Using doSQL() to Execute Imperative SQL Operations

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.

## 5.4.5 Using updater() to Execute an update Statement

The `updater()` method performs a Transact-SQL `update` statement.

The `update` action updates the `name` and `home` columns for all rows with a given `<id>` value:

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

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.

## 5.4.6 Using `selector()` to Execute a select Statement

The `selector( )` method executes a Transact-SQL `select` statement.

The `select` action is:

```
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, `selector( )` supplies a value for the parameter marker after preparing the SQL statement:

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

`selector( )` then executes the `select` statement:

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

### **i** 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.

`selector( )` requires that the `<ResultSet>` object have exactly one row. The `selector( )` 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.

## 5.4.7 Using caller() to Call a SQL Stored Procedure

The `caller()` method calls the stored procedure `inoutproc`.

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>`.

```
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
```

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()`, `updateAction()`, 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);
```

## 5.5 Error Handling in the Native JDBC Driver

SAP ASE 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 SAP ASE 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 to 18) – are converted to `SQLExceptions`.
- Fatal errors (severity 19 to 24) – are converted to fatal `SQLExceptions`.

`SQLExceptions` can be raised through JDBC, SAP ASE, 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. SAP ASE 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 SAP ASE, SAP ASE 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.

### i 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.

## 5.6 The JDBC Examples Class

The example class shows the use of JDBC facilities with Java in SAP ASE. 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 = connector(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 = connector("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 connector( ) 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 connector(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 += selector(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 selector( ) 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 selector(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 + );
}
}

```

# 6 SQLJ Functions and Stored Procedures

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.

## 6.1 Creating a SQLJ Stored Procedure or Function

Create and execute a SQLJ stored procedure or function.

### Procedure

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

Refer to *Preparing for and Maintaining Java in the Database*, for information on creating, compiling, and installing Java methods in SAP ASE .

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.

### Related Information

[Preparing for and Maintaining Java in the Database \[page 27\]](#)

## 6.2 Compliance with SQLJ Part 1 Specifications

SAP ASE SQLJ stored procedures and functions comply with SQLJ Part 1 of the standard specifications for using Java with SQL.

In those instances where SAP proprietary implementation differs from the SQLJ specifications, SAP 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 SAP syntax for creating SQLJ stored procedures supports all three parameter modes.

See *Standards* for a description of the SQLJ standards.

SAP ASE supports most features described in the SQLJ Part 1 specification; however, there are some differences. See, *SQLJ and SAP Implementation: A Comparison*.

### Related Information

[Standards \[page 10\]](#)

## 6.3 Security and Permissions

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

SQLJ functions and user-defined functions (UDFs) 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*.

## Related Information

[Invoking Java Methods in SQL \[page 39\]](#)

## 6.4 SQLJ Examples

The examples provided assume specific table and column names.

The table is 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.

## Related Information

[SQLJExamples Class \[page 108\]](#)

## 6.5 Invoke Java Methods in SAP ASE

You can invoke Java methods directly in SQL, or indirectly using SQLJ stored procedures and functions.

- Invoke Java methods directly in SQL. Directions for invoking methods in this way are presented in *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 SAP ASE database using the `installjava` utility. See *Preparing for and Maintaining Java in the Database*, for more information.

### Related Information

[Using Java Classes in SQL \[page 34\]](#)

[Preparing for and Maintaining Java in the Database \[page 27\]](#)

### 6.5.1 Invoke 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.

### Related Information

[Using Java Classes in SQL \[page 34\]](#)

### 6.5.2 Invoke 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.
- SAP SQLJ conforms to the recognized SQLJ Part 1 standard, which allows you to use SAP SQLJ procedures and functions in conforming non-SAP environments.
- You can invoke SQLJ functions and SQLJ stored procedures across databases.
- Because SAP ASE 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.

## 6.6 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. SAP 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 SAP ASE 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 *Nulls in the Function Call*.
- You can include the `deterministic` or `not deterministic` keywords, but SAP ASE 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 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, SAP ASE infers the Java method signature from the SQL function signature. SAP recommends that you include the method signature as this practice handles all datatype translations. See *Map Java and SQL Datatypes*.
- 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 SAP ASE is:

1. Built-in functions
2. SQLJ functions
3. Java-SQL functions that are called directly

## Related Information

[Nulls in the Function Call \[page 92\]](#)

[Map Java and SQL Datatypes \[page 101\]](#)

### 6.6.1 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 with a datatype that 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";
}
```

## 6.6.1.1 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.

### i 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.

## 6.6.1.2 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`.

## 6.6.2 Delete 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.

## 6.7 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 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 SAP ASE 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>
  ...]])]'
```

### i Note

To comply with the ANSI standard, the SQLJ `create procedure` command syntax is different from syntax used to create SAP 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. SAP 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, SAP ASE does not make use of this option.
- You must include the `language java parameter style java` keywords, which tell SAP ASE 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, SAP ASE infers one from the SQL procedure signature. SAP recommends that you include the method signature as this practice handles all datatype translations. See *Map Java and SQL Datatypes* 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.

## Related Information

[Map Java and SQL Datatypes \[page 101\]](#)

## 6.7.1 Modify 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 SAP 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.setString(1, newSpelling);
```

```

        pstmt.setString(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'

```

## 6.7.2 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 SAP 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; SAP ASE 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.

## i 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_emp WHERE"
                + "region_of(state)>? AND"
                + "sales IS NOT NULL ORDER BY sales DESC");
        stmt.setInt(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

```

### i Note

SAP ASE 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.

## 6.7.3 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.

### 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.

SAP ASE 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. SAP ASE closes them automatically.

### Note

SAP ASE 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 procedure matches only one Java method.

### i Note

Some restrictions apply to method overloading when you infer a method signature involving result sets. See *Map Java and SQL Datatypes* 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.getBigDecimal(3);
    System.out.print("Name = " + name);
    System.out.print("Region = "+ region);
    System.out.print("Sales = "+ sales);
    System.out.println();
}
```

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.

### i Note

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

## Related Information

[Map Java and SQL Datatypes \[page 101\]](#)

## 6.7.3.1 Delete 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.

## 6.8 View 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_helpprotect` 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.

## 6.9 Map 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.

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*.
- A SQL datatype and a non-primitive Java datatype are *object mappable*.
- 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 SAP extension of the SQLJ Part 1 standard.

Table 3: Simply and Object Mappable SQL and Java Datatypes

SQL Datatype	Simply Mappable Java Datatype	Object Mappable Java Datatype
<code>char/unichar</code>		<code>java.lang.String</code>
<code>nchar</code>		<code>java.lang.String</code>
<code>varchar/univarchar</code>		<code>java.lang.String</code>
<code>nvarchar</code>		<code>java.lang.String</code>
<code>text</code>		<code>java.lang.String</code>
<code>numeric</code>		<code>java.math.BigDecimal</code>
<code>decimal</code>		<code>java.math.BigDecimal</code>
<code>money</code>		<code>java.math.BigDecimal</code>
<code>smallmoney</code>		<code>java.math.BigDecimal</code>
<code>bit</code>	<code>boolean</code>	<code>boolean</code>
<code>tinyint</code>	<code>byte</code>	<code>Integer</code>
<code>smallint</code>	<code>short</code>	<code>Integer</code>
<code>integer</code>	<code>int</code>	<code>Integer</code>
<code>bigint</code>	<code>long</code>	<code>java.math.BigInteger</code>
<code>unsigned smallint</code>	<code>int</code>	<code>Integer</code>
<code>unsigned int</code>	<code>long</code>	<code>Integer</code>
<code>unsigned bigint</code>		<code>java.math.BigInteger</code>
<code>real</code>	<code>float</code>	<code>Float</code>
<code>float</code>	<code>double</code>	<code>Double</code>

SQL Datatype	Simply Mappable Java Datatype	Object Mappable Java Datatype
double precision	double	Double
binary		byte[]
varbinary		byte[]
datetime		java.sql.Timestamp
smalldatetime		java.sql.Timestamp
date		java.sql.Date
time		java.sql.Time

## 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, SAP ASE looks for the single method that matches the method name and signature. For example:

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

In this case, SAP ASE resolves parameter types using normal Java overloading conventions.

If you do not specify the Java method signature, however:

```
create procedure ranked_emp (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 SAP ASE 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, SAP ASE 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.

## Related Information

[Null Argument Values \[page 90\]](#)

### 6.9.1 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 SAP ASE 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 the table *Simply and Object Mappable SQL and Java Datatypes*.

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

You can allow SAP ASE to infer the method signature for datatypes that are:

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

For example, if you want SAP ASE 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'
```

SAP ASE 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, SAP ASE infers the primitive, simply mappable datatype.



For example, the `SQLJExamples.job` method contains a parameter of type `int`. When creating a function referencing that method, SAP ASE 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)'
```

## 6.10 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.

## 6.11 SQLJ and SAP Implementation: A Comparison

There are differences between SQLJ Part 1 standard specifications and the SAP proprietary implementation for SQLJ stored procedures and functions.

Table 4: SAP Enhancements

Category	SQLJ Standard	SAP Implementation
<code>create procedure</code> 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 <code>create procedure</code> can have either void or integer return types.
<code>create procedure</code> and <code>create function</code> commands	Supports only SQL datatypes in <code>create procedure</code> or <code>create function</code> 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 OmniConnect capabilities.
<code>drop procedure</code> and <code>drop function</code> commands	Requires complete command name: <code>drop procedure</code> or <code>drop function</code> .	Supports complete function name and abridged names: <code>drop proc</code> and <code>drop func</code> .

Table 5: SQLJ Features not Supported

SQLJ Category	SQLJ Standard	SAP Implementation
<code>create function</code> command	Allows users to specify the same SQL name for multiple SQLJ functions.	Requires unique names for all stored procedure and functions.
utilities	Supports <code>sqlj.install_jar</code> , <code>sqlj.replace_jar</code> , <code>sqlj.remove_jar</code> , and similar utilities to install, replace, and remove JAR files.	Supports the <code>installjava</code> utility and the <code>remove java</code> Transact-SQL command to perform similar functions.

Table 6: SQLJ Features Partially Supported

SQLJ Category	SQLJ Standard	SAP Implementation
<code>create procedure</code> and <code>create function</code> 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.

SQLJ Category	SQLJ Standard	SAP Implementation
<code>create procedure</code> and <code>create function</code> commands	Supports the key words <code>no sql</code> , <code>contains sql</code> , <code>reads sql</code> data, and <code>modifies sql data</code> to specify the SQL operations the Java method can perform.	Supports <code>modifies sql data</code> only.
<code>create procedure</code> command	Supports <code>java.sql.ResultSet</code> and the SQL/OLB iterator declaration.	Supports <code>java.sql.ResultSet</code> only.
<code>drop procedure</code> and <code>drop function</code> commands	Supports the key word <code>restrict</code> , 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 <code>restrict</code> key word and functionality.

Table 7: SQLJ Features Defined by the Implementation

SQLJ Category	SQLJ Standard	SAP Implementation
<code>create procedure</code> and <code>create function</code> commands	Supports the <code>deterministic   not deterministic</code> keywords, which specify whether or not the procedure or function always returns the same values for the <code>out</code> and <code>inout</code> parameters and the function result.	Supports only the syntax for <code>deterministic   not deterministic</code> , not the functionality.
<code>create procedure</code> and <code>create function</code> commands	The validation of the mapping between the SQL signature and the Java method signature can be performed either when the <code>create</code> 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 <code>create</code> command is executed, which enables faster execution.
<code>create procedure</code> and <code>create function</code> commands	Can specify the <code>create procedure</code> or <code>create function</code> commands within deployment descriptor files or as SQL DDL statements. The implementation defines which way (or ways) the commands are supported.	Supports <code>create procedure</code> and <code>create function</code> as SQL DDL statements outside of deployment descriptors.

SQLJ Category	SQLJ Standard	SAP Implementation
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 <code>Exception.sqlException</code> subclass. The effect of the exception condition is defined by the implementation.	Follows the rules for SAP ASE 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.

## 6.12 SQLJExamples Class

The `SQLJExamples` class is 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.setInt(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;  
}  
}
```

# 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 SAP ASE.

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 SAP ASE database is done remotely.

## 7.1 Setting up Java Debugging

Whether you use an IDE, a standalone debugger, or a jdb debugger, you must configure the server to support debugging and attach the remote debugger to the JVM debugging agent.

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.

### **i** 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.

### 7.1.1 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"
```

### **i** Note

Use `pca_jvm_java_dbg_agent_suspend` with caution. Enabling `pca_jvm_java_dbg_agent_suspend` causes the JVM to suspend and all SAP ASE Java tasks to wait until you attach and instruct the JVM to continue via the debugger. SAP 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).

### **i** 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.

## 7.1.2 Attaching the Remote Debugger to the JVM Debug Agent

A debug session begins when the remote debugger attaches to the debug agent running in SAP ASE.

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.

# 8 File and Network Access Using Java

SAP ASE supports both file and network I/O capabilities using `java.io`, `java.net`, and `java.nio` packages.

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.

## Related Information

[The PCI Memory Pool \[page 18\]](#)

## 8.1 Accessing Files 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.

Make sure there is a clear distinction between the user identity used by the operating system and the user identity used by SAP ASE.

### 8.1.1 User Identity and Permissions

When SAP ASE starts, the server process executes using the system user ID that started the process.

For example, if SAP ASE 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 SAP ASE process and the operating system are associated with the system user ID that started SAP ASE.

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 SAP ASE 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 SAP ASE 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 SAP ASE 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 SAP ASE 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 SAP ASE 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.

## 8.1.2 Specifying Directories for File I/O (UNIX)

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.

### 8.1.2.1 Mask Syntax

The `work_dir` permission mask must follow certain conventions.

`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]ll 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"
```

## 8.1.3 Specifying Directories for File I/O (Windows)

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

Use 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.

## 8.1.4 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

## 8.1.5 Rules for Opening Existing Files

Rules and checks apply for opening files on UNIX and Windows platforms.

### i 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.

### i 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. SAP ASE 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.

## 8.1.6 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.

### i 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.

## 8.1.7 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.

## 8.2 File Access Using `java.net`

SAP ASE 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 SAP ASE 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.

### i 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.

## 8.2.1 Examples for Socket Classes and the URL Class

Examples are provided 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 `SAP ASE` memory and creates a new `InputStream` on the document in memory.

```
import java.io.*;
import java.net.*;
public class URLprosess {
    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 SAP ASE , 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.println("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 (c1 text)
```

```
go
insert into mytable values (URLprocess.readURL())
go
Number of bytes read :40867
select datalength(c1) from mytable
go
```

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

## 9 Suggestions for Improving Performance

There are guidelines for improving performance when using Java in SAP ASE.

### 9.1 Minimize the Number of Calls from SQL to the 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 Address
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.

## 9.2 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 SAP ASE.

Threads created within a Java method compete with SAP ASE for CPU and other resources. Large numbers or resource-intensive threads can impact overall server performance.

## 9.3 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 SAP `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
    ...
}
```

## 9.4 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: Write loop in Java, and call methods from Java-coded loop.

# 10 Unsupported Java API Packages, Classes, and Methods

SAP ASE supports many but not all classes and methods in the Java API. In addition, SAP ASE may impose security restrictions and implementation limitations.

For example, SAP ASE does not support all of the thread manipulation facilities of `java.lang.Thread`.

## ⚠ Caution

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 SAP ASE 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 SAP ASE does not support the native methods invoked through the Java Native Interface (JNI).

This section lists:

- Unsupported Java methods
- Unsupported `java.sql` methods

## 10.1 Restricted Java Packages, Classes, and Methods

There are considerations for 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()`
  - `getAllStackTraces()`

- `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.

## 10.2 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 SAP 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()`



# 11 Java-SQL Reference Information

## 11.1 Java-SQL Identifiers

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

Type	Usage
Java-SQL Identifiers	<ul style="list-style-type: none"><li>• Java-SQL identifiers can be up to 255 bytes long if they are surrounded by quotation marks. Otherwise, they are 30 bytes or fewer.</li><li>• The first character of the identifier is 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.</li><li>• Java-SQL identifiers are always case sensitive.</li></ul>
Delimited Identifiers	<ul style="list-style-type: none"><li>• 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.</li></ul>

#### i Note

You can use double quotes with Java-SQL identifiers whether the `set quoted_identifier` option is `on` or `off`.

## Type

## Usage

- 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.

## 11.2 Java-SQL Class and Package Names

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, SAP ASE 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.

## Related Information

[Java-SQL Identifiers \[page 129\]](#)

## 11.3 Java-SQL Column Declarations

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

## 11.4 Java-SQL Variable Declarations

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.

## 11.5 Java-SQL Column References

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 following is the value of the column reference, where CSC is the column reference:

```
CSC . readObject (S)
```

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.

## 11.6 Java-SQL Member References

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.

## 11.7 Java-SQL Method Calls

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

- When there are methods with the same name in the same class or instance, the issue is resolved according to Java method overloading rules.
- 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.
- 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.
  - SAP ASE does not pass the null value as the value of a parameter to a method whose Java type is scalar.



# 12 Glossary

This glossary describes Java and Java-SQL terms used in this book.

## i Note

For a description of SAP ASE and SQL terms, refer to the *SAP ASE Glossary*.

<b>assignment</b>	A generic term for the data transfers specified by <code>select</code> , <code>fetch</code> , <code>insert</code> , and <code>update</code> Transact-SQL commands. An assignment sets a source value into a target data item.
<b>associated JAR</b>	If a class/JAR is installed with <code>installjava</code> and the <code>-jar</code> option, then the JAR is retained in the database and the class is linked in the database with the associated JAR. See <a href="#">retained JAR [page 139]</a> .
<b>bytecode</b>	The compiled form of Java source code that is executed by the Java VM.
<b>class</b>	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 <a href="#">class instance [page 137]</a> .
<b>class method</b>	See <a href="#">static method [page 139]</a> .
<b>class file</b>	A file of type "class" (for example, <code>myclass.class</code> ) that contains the compiled bytecode for a Java class. See <a href="#">Java file [page 138]</a> and <a href="#">Java archive (JAR) [page 138]</a> .
<b>class instance</b>	Value of the class data type that contains a value for each field of the class and that accepts all methods of the class.
<b>datatype mapping</b>	Conversions between Java and SQL datatypes.
<b>declared class</b>	The declared datatype of a Java-SQL data item. It is either the datatype of the runtime value or a supertype of it.
<b>externalization</b>	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 <a href="#">serialization [page 139]</a> .
<b>installed classes</b>	Java classes and methods that have been placed in the SAP ASE system by the <code>installjava</code> utility.
<b>instance method</b>	A invoked method that references a specific instance of a class.

<b>interface</b>	A named collection of method declarations. A class can implement an interface if the class defines all methods declared in the interface.
<b>Java archive (JAR)</b>	A platform-independent format for collecting classes in a single file.
<b>Java Database Connectivity (JDBC)</b>	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.
<b>Java datatypes</b>	Java classes, either user-defined or from the JavaSoft API, or Java primitive datatypes, such as <code>boolean</code> , <code>byte</code> , <code>short</code> , and <code>int</code> .
<b>Java Development Kit (JDK)</b>	A toolset from Sun Microsystems that allows you to write and test Java programs from the operating system.
<b>Java file</b>	A file of type "java" (for example, <code>myfile.java</code> ) that contains Java source code. See <a href="#">class file [page 137]</a> and <a href="#">Java archive (JAR) [page 138]</a> .
<b>Java method signature</b>	The Java datatype of each parameter of a Java method.
<b>Java object</b>	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.
<b>Java-SQL column</b>	A SQL column whose datatype is a Java-SQL class.
<b>Java-SQL class</b>	A public Java class that has been installed in the SAP ASE 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.
<b>Java-SQL datatype mapping</b>	Conversions between Java and SQL datatypes. See <a href="#">Datatype Mapping Between Java and SQL [page 51]</a> .
<b>Java-SQL variable</b>	A SQL variable whose datatype is a Java-SQL class.
<b>Java Virtual Machine (JVM)</b>	The Java interpreter that processes Java in the server. It is invoked by the SQL implementation.
<b>mappable</b>	<p>A Java datatype is mappable if it is either:</p> <ul style="list-style-type: none"> <li>• Listed in the first column of <a href="#">Table: Mapping SQL datatypes to Java types [page 51]</a>, or</li> <li>• A public Java-SQL class that is installed in the SAP ASE system.</li> </ul> <p>A SQL datatype is mappable if it is either:</p> <ul style="list-style-type: none"> <li>• Listed in the first column of <a href="#">Table: Mapping Java scalar types to SQL datatypes [page 52]</a>, or</li> <li>• A public Java-SQL class that is built-in or installed in the SAP ASE system.</li> </ul> <p>A Java method is mappable if all of its parameter and result datatypes are mappable.</p>

<b>method</b>	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.
<b>narrowing conversion</b>	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 <code>convert</code> function. See also <a href="#">widening conversion [page 140]</a> .
<b>package</b>	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 <code>import</code> statements to specify other packages whose classes can then be referenced.
<b>pluggable component adaptor/ JVM</b>	A SAP component that manages service requests between SAP ASE and the JVM.
<b>pluggable component interface (PCI)</b>	The SAP ASE Java framework, which lets you, with the help of the PCA/JVM, use a commercially available JVM with SAP ASE .
<b>pluggable component interface (PCI) Bridge</b>	An SAP ASE component, and part of the PCI, that enables interaction between the JVM plug-in and SAP ASE .
<b>procedure</b>	An SQL stored procedure, or a Java method with a <i>void</i> result type.
<b>public</b>	Public fields and methods, as defined in Java.
<b>retained JAR</b>	See <a href="#">associated JAR [page 137]</a> .
<b>serialization</b>	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 <a href="#">externalization [page 137]</a> .
<b>SQL function signature</b>	The SQL datatype of each parameter of a SQLJ function.
<b>SQL-Java datatype mapping</b>	Conversions between Java and SQL datatypes. See <a href="#">Datatype Mapping Between Java and SQL [page 51]</a> .
<b>SQL procedure signature</b>	The SQL datatype of each parameter of a SQLJ procedure.
<b>static method</b>	A method invoked without referencing an object. Static methods affect the whole class, not an instance of the class. Also called a class method.
<b>subclass</b>	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 \[page 140\]](#), [narrowing conversion \[page 139\]](#), and [widening conversion \[page 140\]](#).



- 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 \[page 139\]](#), [narrowing conversion \[page 139\]](#), and [widening conversion \[page 140\]](#).
- 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 \[page 139\]](#).

# Important Disclaimers and Legal Information

## Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.

About the icons:

- Links with the icon : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon : You are leaving the documentation for that particular SAP product or service and are entering a SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

## Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.

The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

## Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

## Gender-Related Language

We try not to use gender-specific word forms and formulations. As appropriate for context and readability, SAP may use masculine word forms to refer to all genders.

© 2019 SAP SE or an SAP affiliate company. All rights reserved.

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

Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see <https://www.sap.com/about/legal/trademark.html> for additional trademark information and notices.