CHAPTER 16

Welcome to Java in the Database

About this chapter

This chapter provides motivation and concepts for using Java in the database.

Adaptive Server Anywhere is a runtime environment for Java, or Java platform. Java provides a natural extension to SQL, turning Adaptive Server Anywhere into a platform for the next generation of enterprise applications.

Contents

Topic	Page
Introduction to Java in the database	430
Java in the database Q & A	433
A Java seminar	439
The Sybase runtime environment for Java	448
A Java in the database exercise	458

Introduction to Java in the database

Adaptive Server Anywhere is a **runtime environment for Java**. This means that Java classes can be executed in the database server. Building a runtime environment for Java classes into the database server provides powerful new ways of managing and storing data and logic.

Java in the database offers the following:

- ♦ You can reuse Java components in the different layers of your application—client, middle-tier, or server—and use them wherever makes most sense to you. Adaptive Server Anywhere becomes a platform for distributed computing.
- ♦ 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.

The SQLJ proposed standard

The Adaptive Server Anywhere Java implementation is based on the SQLJ Part 1 and SQLJ Part 2 proposed standards. SQLJ Part 1 provides specifications for calling Java static methods as SQL stored procedures and user-defined functions SQLJ Part 2 provides specifications for using Java classes as SQL user-defined data types.

Learning about Java in the database

Java is a relatively new programming language with a growing but still limited knowledge base. This documentation is written not only for experienced Java developers, but also for the many readers who are unfamiliar with the language, its possibilities, its syntax and its use.

For those readers familiar with Java, there is much to learn about how to use Java in a database. Sybase is not only extending the capabilities of the database with Java, but also extending the capabilities of Java with the database.

Java documentation

The following table outlines the documentation regarding the use of Java in the database.

Title	Purpose
"Welcome to Java in the Database" on page 429 (this chapter)	Java concepts and how they are applied in Adaptive Server Anywhere.
"Using Java in the Database" on page 465	Practical steps to using Java in the database.
"Data Access Using JDBC" on page 503	Accessing data from Java classes, including distributed computing.
"Debugging Java in the Database" on page 533	Testing and debugging Java code running in the database.
Adaptive Server Anywhere Reference.	The *Reference Manual* includes material on the SQL extensions that support Java in the database.
Reference guide to Sun's Java API	Online guide to Java API classes, fields and methods. Available as Windows Help only.
Thinking in Java by Bruce Eckel.	Online book that teaches how to program in Java. Supplied in Adobe PDF format in the *jxmp* subdirectory of your Adaptive Server Anywhere installation directory.

Using the Java documentation

The following table is a guide to which parts of the Java documentation apply to you, depending on your interests and background. It is meant to act only as a guide and should not limit your efforts to learn more about Java in the database.

If you ...	Consider reading ...
Are new to object-oriented programming.	"A Java seminar" on page 439
	Thinking in Java by Bruce Eckel.
Want an explanation of terms such as **instantiated**, **field** and **class method**.	"A Java seminar" on page 439
Are a Java developer who wants to just get started.	"The Sybase runtime environment for Java" on page 448
	"A Java in the database exercise" on page 458
Want to know the key features of Java in the database.	"Java in the database Q & A" on page 433

If you ...	Consider reading ...
Want to find out how to access data from Java.	"Data Access Using JDBC" on page 503
Want to prepare a database for Java.	"Java-enabling a database" on page 469
Want a complete list of supported Java APIs.	"Java class data types" on page 243 of the book *Adaptive Server Anywhere Reference Manual*
Are trying to use a Java API class and need Java reference information.	The online guide to Java API classes (Windows Help only).
Want to see an example of distributed computing.	"Creating distributed applications" on page 527.

Java in the database Q & A

This section describes the key features of Java in Adaptive Server Anywhere.

What are the key features of Java in the database?

All the following points are explained in detail in later sections.

- ♦ **You can run Java in the database server** An internal Java Virtual Machine (VM) runs Java code in the database server.
- ♦ **You can call Java from SQL** You can call Java functions (**methods**) from SQL statements. Java methods provide a more powerful language than SQL stored procedures for adding logic to the database.
- ♦ **You can access data from Java** An internal JDBC driver lets you access data from Java.
- ♦ **You can debug Java in the database** You can use the Sybase Java debugger to test and debug your Java classes in the database.
- ♦ **You can use Java classes as data types** Every Java class installed in a database becomes available as a data type that can be used as the data type of a column in a table or a variable.
- ♦ **You can save Java objects in tables** An instance of a Java class (a Java object) can be saved as a value in a table. Java objects can be inserted into a table, SELECT statements can be executed against the fields and methods of objects stored in a table, and Java objects can be retrieved from a table.

  With this ability, Adaptive Server Anywhere becomes an object-relational database, supporting objects while not degrading existing relational functionality.
- ♦ **SQL is preserved** The use of Java does not alter the behavior of existing SQL statements or other aspects of non-Java relational database behavior.

How do I store Java instructions in the database?

Java is an object-oriented language, so its instructions (source code) come in the form of classes. In order to execute Java in a database, you write the Java instructions outside the database, and compile them outside the database into compiled classes (**byte code**), which are binary files holding Java instructions.

You then **install** these compiled classes into a database. Once installed, these classes can be executed in the database server.

Adaptive Server Anywhere is a runtime environment for Java classes, not a Java development environment. You need a Java development environment, such as Sybase PowerJ or the Sun Microsystems Java Development Kit, to write and compile Java.

How does Java get executed in a database?

Adaptive Server Anywhere includes a **Java Virtual Machine** (**VM**), which runs in the database environment. The Sybase Java VM interprets compiled Java instructions and runs them in the database server.

The Sybase Java VM supports public class and instance methods; classes inheriting from other classes; packages; the Java API; and access to protected, public and private fields. Some Java API functions that are not appropriate in a server environment are not supported, including user interface elements.

In addition to the VM, the SQL request processor in the database server has been extended so that it can call into the VM to execute Java instructions. It can also process requests from the VM, to enable data access from Java.

Differences from a standalone VM

There is a difference between executing Java code using a standard VM such as the Sun Java VM *java.exe* and executing Java code in the database. The Sun VM is run from a command line, while the Adaptive Server Anywhere Java VM is available at all times to perform a Java operation whenever it is required as part of the execution of a SQL statement.

The Sybase Java interpreter cannot be accessed externally. It is only used when the execution of a SQL statement requires a Java operation to take place. The database server starts the VM automatically when needed: you do not have to take any explicit action to start or stop the VM.

Why Java?

Java provides a number of features that make it ideal for use in the database:

- Thorough error checking at compile time.
- Built-in error handing with a well-defined error handling methodology.
- Built-in garbage collection (memory recovery).
- Elimination of many bug-prone programming techniques.
- Strong security features

- ♦ Java code is interpreted, so no operations get executed without being acceptable to the VM.

How do I use Java and SQL together?

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

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

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

What is the Java API?

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

The Java API offers a tremendous amount of functionality in its own right. A large portion of the Java API is built in to any database that is enabled to use Java code. This exposes the majority of non-visual classes from the Java API that should be familiar to developers currently using the Sun Microsystems Java Development Kit (JDK).

For a complete list of supported Java APIs, see "Built-in Java classes" on page 243 of the book *Adaptive Server Anywhere Reference Manual*.

How do I access the Java API from SQL?

In addition to using the Java API in classes, it can be used in stored procedures and SQL statements. You can treat the Java API classes as extensions to the available built-in functions provided by SQL.

For example, the SQL function PI(*) returns the value for Pi. The Java API class `java.lang.Math` has a parallel field named PI returning the same value. But `java.lang.Math` also has a field named E that returns the base of the natural logarithms, as well as a method that computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.

Other members of the Java API offer even more specialized functionality. For example, `java.util.Stack` generates a last-in, first-out queue that can store a list in an ordered manner; `java.util.HashTable` is used to map values to keys; `java.util.StringTokenizer` breaks a string of characters into individual word units.

Which Java classes are supported?

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

For a list of supported and unsupported classes, see "Built-in Java classes" on page 243 of the book *Adaptive Server Anywhere Reference Manual*, and "Unsupported packages and classes" on page 250 of the book *Adaptive Server Anywhere Reference Manual*.

How can I use my own Java classes in databases?

You can install your own Java classes into a database. For example a user-created Employee class or Package class that a developer designed, wrote in Java, and compiled with a Java compiler.

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

Java classes not the same as stored procedures
This is not the same as writing stored procedures in Java. Stored procedures are written in SQL. Java classes provide a more powerful language than stored procedures, and yet can be called from client applications in the same way that stored procedures are called.

When a Java class gets installed in a database, it becomes available as a new user-defined data type. A Java class can be used in any situation where built-in SQL data types can be used: as a column type in a table or a variable type.

For example, if a class called `Address` has been installed into a database, a column in a table called **Addr** can be of type **Address**, which means only objects based on the `Address` class can be saved as row values for that column.

Can I access data using Java?

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

Normally, JDBC classes are used within a client application, and the database system vendor supplies a JDBC driver that allows the JDBC classes to establish a connection.

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

Can I move classes from client to server?

This design allows you to create Java classes that can be moved between levels of an enterprise application: the same Java class can be integrated into either the client application, a middle tier, or the database—wherever it is most appropriate.

A class that contains business logic, data or a combination of both, can be moved to any level of the enterprise system, including the server. This allows you complete flexibility to make the most appropriate use of resources.

This enables enterprise customers to develop their applications using a single programming language in a multi-tier architecture with unparalleled flexibility.

Can I create distributed applications?

You can create an application that has some pieces operating in the database and some on the client machine. You can pass Java objects from the server to the client. This includes objects stored in database tables.

☞ For an example, see "Creating distributed applications" on page 527.

What can I *not* do with Java in the database?

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

You cannot carry out the following tasks in the database:

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

The Java classes used in Adaptive Server Anywhere must be written and compiled using a Java application development tool, and then installed into a database for use, testing, and debugging.

A Java seminar

This section introduces key Java concepts. After reading this section you should be able to examine Java code, such as a simple class definition or the invocation of a method, and understand what is taking place.

Java examples directory
All complete classes used as examples in this document are located in the Java examples directory, *jxmp*, which is a sub directory of your Adaptive Server Anywhere installation directory.

Each Java class example is represented by two files: the Java class declaration and the compiled version. The compiled version of the Java class examples can be immediately installed to a database without modification.

Object oriented and procedural languages

If you are more familiar with procedural languages such as C, or the SQL stored procedure language, than object-oriented languages, this section explains some of the key similarities and differences between procedural and object-oriented languages.

Java is based on classes

The main structural unit of code in Java is a **class**.

A Java class could be looked at as just a collection of procedures and variables that have been grouped together because they all relate to a specific, identifiable category.

However the manner in which a class gets used sets object oriented languages apart from procedural languages. When an application written in a procedural language is executed, it is typically loaded into memory once and takes the user down a pre-defined course of execution.

In object-oriented languages such as Java, a class is used like a template: a definition of potential program execution. Multiple copies of the class can be created and loaded dynamically, as needed, with each **instance** of the class capable of containing its own data, values and course of execution. Each loaded class could be acted on or executed independently of any other class loaded into memory.

A class that is loaded into memory for execution it is said to have been **instantiated**. An instantiated class is called an object: it is an application derived from the class that is prepared to hold unique values or have its methods executed in a manner independent of other class instances.

Understanding the Java class

One way of understanding the concept of a class is to view it as an entity, an abstract representation of a thing.

An Invoice class, for example, could be designed to mimic a paper invoice such as those used every day in business operations. A paper invoice contains certain information, like line-item details, who is being invoiced, the date of the invoice, the payment amount and when payment is due. This is information that could also be contained in an instance of an Invoice class.

In addition to just holding data, a class is capable of calculations and logical operations. The Invoice class, for example, could be designed to calculate the tax on a list of line items, and add it to the sub total to produce a final total. This could be done for every Invoice object without user intervention.

Such a class could also ensure all essential pieces of information are added to the Invoice and even indicate when payment is over due or partially paid.

A class combines data and functionality: the ability to hold information and perform computational operations.

Example

The following Java code declares a class called Invoice. This class declaration would be stored in a file named *Invoice.java*. It could then be compiled into a Java class, using a Java compiler.

A note about compiling Java classes
Compiling a Java class declaration does not alter the class declaration: it creates a new file with the same name but a different extension. Compiling *Invoice.java* creates a file called *Invoice.class* that could be used in a Java application and executed by a Java VM.

The Sun JDK tool for compiling class declarations is *javac.exe*.

```
public class Invoice {
  // So far, this class does nothing and knows nothing
}
```

The **`class`** keyword is used, followed by the name of the class. There is an opening and closing brace: everything declared between the braces, such as fields and methods, becomes a part of the class.

In fact, no Java code exists outside class declarations. Even the Java procedure that a Java interpreter runs automatically to create and manages other objects — the **`main`** method that is often the start of your application — is itself located within a class declaration.

Java classes and objects

A **class** is the template for what can be done, just as an invoice form is only the template used to generate an invoice. A class defines what an object is capable of doing just as an invoice form defines what information the invoice should contain.

Information is not held in the class, an object is created based on the class and the object is used to hold data or perform calculations or other operations.

An **object** is said to be of **type** *JavaClass*, where *JavaClass* is the name of the class upon which the object is based. An Invoice object is an instance of the Invoice class. The class defines what the object is capable of but the object is the incarnation of the class that gives the class meaning and usefulness.

This is similar to the invoice example. The invoice form defines what all invoices based on that form can accomplish. There is one form and zero or many invoices based on the form. The form contains the definition but the invoice does the work.

Similarly it is the Invoice object that gets created, stores information, is stored, retrieved, edited, updated, and so on.

Just as one invoice template is used to create many invoices, with each invoice separate and distinct from the other in its details, many objects can be generated from one class.

Methods and fields

A **method** is the part of the class that does something — a function that performs a calculation or interacts with other objects — on behalf of the class. Methods can accept arguments, and return a value to the calling function. If no return value is needed, a method can return **void**. Classes can have any number of methods.

A **field** is the part of class that ends up holding information. When an object of type *JavaClass* is created, the fields in *JavaClass* are available to be passed values unique to that object.

Examples

The following declaration of the class Invoice has four fields, corresponding to information that might be contained on two line items on an invoice.

To declare a field in a class, state its type and then its name, followed by a semicolon. Such a variable is a field if it is declared in the body of the class and not within a method. (Declaring a variable within a method makes it a part of the method, not the class.)

```
public class Invoice {

    // Fields are things an Invoice object knows
    public String lineItem1Description;
```

```
        public int lineItem1Cost;

        public String lineItem2Description;
        public int lineItem2Cost;

    }
```

Adding methods

Another possible modification to the Invoice class is to include a method. A method is declared by stating its return type, its name and what parameters it takes (in this case, none). Like a class declaration, the method uses an opening and closing brace to identify the body of the method where the code goes.

```
public class Invoice {

    // Fields
    public String lineItem1Description;
    public double lineItem1Cost;

    public String lineItem2Description;
    public double lineItem2Cost;

    // A method
    public double totalSum() {
        double runningsum;

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * 1.15;

        return runningsum;
    }
}
```

Within the body of the **`totalSum`** method, a variable named **`runningsum`** is declared. This is used first to hold the sub total of the first and second line item cost. This sub total is then multiplied by 15 per cent (the rate of taxation) to determine the total sum.

The local variable (as it is known within the method body) is then returned to the calling function. When the **`totalSum`** method is invoked, it returns the sum of the two line item cost fields plus the cost of tax on those two items.

Instance methods and class methods

Most methods are used in association with an instance of a class. A `totalSum` method in the Invoice class could calculate and add the tax, and return the sum of all costs, but would only be useful if it is called in conjunction with an `Invoice` object, one that had values for its line item costs. The calculation can only be performed for an object, since the object, not the class, contains the line items of the invoice. The class only defines the capability of the object to have line items. Only the object has the data needed to perform such a calculation.

Java methods are divided into two categories:

- **Instance methods** The `totalSum` method just added is an example of an instance method, one that can only be used in a conjunction with an object and object data
- **Class methods** Class methods are also called **static methods**. A class method can be invoked without the need to first create an object. Only the name of the class and method is required to invoke a class method.

  Similar to instance methods, class methods accept arguments and return values. Typically, class methods perform some sort of utility or information function related to the overall functionality of the class.

The `parseInt` method of the `java.lang.Integer` class, which is supplied with Adaptive Server Anywhere, is one example of a class method. When given a string argument, the `parseInt` method returns the integer version of the string.

For example given the string value "1", the `parseInt` method returns 1, the integer value, without requiring an instance of the `java.lang.Integer` class to first be created as illustrated by this Java code fragment.

```
String num = "1";
int i = java.lang.Integer.parseInt( num );
```

Fields can also be declared using the `static` Java keyword, which makes them into class fields. A static class variable is like a global variable in procedural languages, except that its value is pre-determined and cannot be changed. A field whose name is all in capital letters is often a static variable (class field).

Examples

The following version of the Invoice class now includes both an instance method and a class method. The class method named `rateOfTaxation` returns the rate of taxation used by the class to calculate the total sum of the invoice.

The advantage of making the **`rateOfTaxation`** method a class method (as opposed to an instance method or field) is that other classes and procedures can use the value returned by this method without having to create an instance of the class first. Only the name of the class and method is required to return the rate of taxation used by this class.

Making it a method, as opposed to a field, allows the application developer to change how the rate is calculated without adversely effecting any objects, applications or procedures that use its return value. Future versions of Invoice could make the return value of the **`rateOfTaxation`** class method based on a more complicated calculation without affecting other methods that use its return value.

```
public class Invoice {

    // Fields
    public String lineItem1Description;
    public double lineItem1Cost;

    public String lineItem2Description;
    public double lineItem2Cost;

    // An instance method
    public double totalSum() {
        double runningsum;
        double taxfactor = 1 + Invoice.rateOfTaxation();

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * taxfactor;

        return runningsum;
    }

    // A class method
    public static double rateOfTaxation() {
        double rate;
        rate = .15;

        return rate;
    }
}
```

A Java glossary

The following items outline some of the details regarding Java classes. It is by no means an exhaustive source of knowledge about the Java language but may aid in the use of Java classes in Adaptive Server Anywhere.

For a thorough examination of the Java language, see the online book *Thinking in Java*, by Bruce Eckel, included with Adaptive Server Anywhere in the file *jxmp\Tjava.pdf*.

Public versus private

The visibility of a field, method or class to other Java objects and operations is determined by what is known as an access modifier — essentially the **`public`**, **`private`** or **`protected`** keyword used in front of any declaration.

Fields can be declared private or public, meaning, respectively, their values are available to code within the object, or to code/classes/objects both inside and outside the object.

Methods can also be declared private or public. This has the same effect as if it were a field being declared private or public.

Fields or methods declared as private cannot be manipulated or accessed by methods outside the class.

Public fields or methods can be directly accessed by other classes and methods. Public fields and methods represent everything the external users of the class (including other classes) need to know or are allowed to know.

Packages and the protected modifier

Protected fields or methods are accessible only to the following:

- ♦ Within their class
- ♦ Within subclasses that inherit from the their class.
- ♦ Within the package of which the class is a part.

A package is a grouping of classes that share a common purpose or category. One member of a package has special privileges to access data and methods in other members of the package, hence the **`protected`** access modifier.

A package is the Java equivalent of a library. It is a collection of classes, which can be made available using the **`import`** statement. The following Java statement imports the utility library from the Java API:

```
import java.util.*
```

Packages are typically held in Jar files, which have the extension *.jar* or *.zip*.

Constructors

A **constructor** is a special method of a Java class that creates an instance of the class and returns a reference to the newly-created Java object.

Classes can define their own constructors, including multiple, overriding constructors. Which constructor is used is determined by which arguments were used in the attempt to create the object. When the type, number and order of arguments used to create an instance of the class match one of the class's constructors, that constructor is used to create the object.

Destructors

There is no such thing as a destructor method in Java (as there is in C++). Java classes can define their own `finalize` method for clean up operations when an object is being discarded, but there is no guarantee that this method will get called.

An object that has no references to it will be removed automatically by a "garbage collection" process. This does not apply to objects stored as values in a table.

Other C++ differences

Everything related to a class is contained within the boundaries of the class declaration, including all methods and fields.

Classes can inherit only from one class. Java uses **interfaces** instead of multiple-inheritance. A class can implement multiple interfaces, each interface defines a set of methods and method profiles that must be implemented by the class in order for the class to be compiled.

An interface is similar to an abstract class: it defines what methods and static fields the class must declare. The implementation of the methods and fields declared in an interface is located within the class that uses the interface: the interface defines what the class must declare, it is up to the class to determine how it is implemented.

Java error handling

Java error handling code is separate from the code for normal processing.

When an error occurs, an exception object representing the error is generated. This is called throwing an **exception**. A thrown exception will terminate a Java program unless it is caught and handled properly at some level of the application.

Both Java API classes and custom-created classes can throw exceptions. In fact, users can create their own exceptions classes, which can be thrown by their own custom-created classes.

If there is no exception handler in the body of the method where the exception occurred, then the search for an exception handler continues up the call stack. If the top of the call stack is reached and no exception handler has been found, the default exception handler of the Java interpreter running the application is called and the program terminates.

In Adaptive Server Anywhere, if a SQL statement calls a Java method, and an unhandled exception is thrown, a SQL error is generated.

Error types in Java

All errors in Java are derived from two types of error classes: **`Exception`** and **`Error`**. Usually, Exception-based errors are handled by error handling code in your method body. Error type errors are reserved for internal errors and resource exhaustion errors inside the Java run-time system.

Exception class errors are thrown and caught. Exception handling code is characterized by **`try`**, **`catch`** and **`finally`** code blocks.

A **`try`** block executes code that may generate an error. A **`catch`** block is code that will execute if an error is generated (thrown) during the execution of a **`try`** block.

A **`finally`** block defines a block of code that executes regardless of whether an error was generated and caught and is typically used for cleanup operations. It is used for code that, under no circumstances, can be omitted.

Exception class errors are divided into two types: those that are runtime exceptions and those that are not runtime exceptions.

Errors generated by the runtime system are known as implicit exceptions, in that they do not have to be explicitly handled as part of every class or method declaration.

For example an array out of bounds exception can occur whenever an array is used, but the error does not have to be part of the declaration of the class or method that uses the array.

All other exceptions are considered to be explicit. If the method being invoked can throw an error, it must be explicitly caught by the class using the exception throwing method, or this class must explicitly throw the error itself by identifying the exception it may generate in its class declaration.

Essentially, explicit exceptions must be dealt with explicitly. A method must declare all the explicit errors it throws, or catch all the explicit errors that may be potentially thrown.

Non-runtime exceptions are checked at compile time. Runtime exceptions are usually caused by errors in programming. Java catches many of such errors during compilation, before the code is run.

Every Java method is given an alternative path of execution so that all Java methods complete, even if they are unable to complete normally. If the type of error that is thrown is not caught, it's passed to the next code block or method in the stack.

The Sybase runtime environment for Java

This section describes the Sybase runtime environment for Java, and how it differs from a standard Java runtime environment.

Java version

The Sybase Java VM executes a subset of JDK version 1.1.6.

Between release 1.0 of the Java Developer's Kit (JDK) and release 1.1, several new APIs where introduced and a number were **deprecated**, that is, the use of certain APIs became no longer recommended and support for them may be dropped in future releases.

A Java class file that uses deprecated APIs generates a warning when it is compiled, but does still execute on a Java virtual machine built to release 1.1 standards, such as the Sybase VM.

The internal JDBC driver supports JDBC version 1.1.

👓 For more information on the JDK 1.1 APIs that are supported, please see "Built-in Java classes" on page 243 of the book *Adaptive Server Anywhere Reference Manual*.

Java classes in the database

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

- The Sybase runtime Java classes.
- User-defined classes
- Table values

Sybase runtime Java classes

The Sybase runtime Java classes are the low-level classes installed to Java-enable a database. These classes include a subset of the Java API.

Like all Java code, Java API functions take the form of classes. These classes are specialized in that, at a low level, they have functionality that no user-defined class could recreate. The Java API is always available to classes in the database.

The Java API classes can be used as pre-defined functions that perform specialized tasks.

Another way to use the Java APIs is to incorporate them in user-created classes: either inheriting their functionality from a Java API class or using it within a calculation or operation in a method.

Some API classes

Some Java API classes of interest include:

- **Primitive Java data types** All primitive (native) data types in Java have a corresponding class. In addition to being able to create objects of these types, the classes have additional, often useful functionality.

  The Java `integer` data type has a corresponding class in `java.lang.Integer`.

- **The utility package** The package `java.util.*` contains a number of very helpful classes whose functionality has no parallel in the SQL functions available in Adaptive Server Anywhere.

  Some of the classes include:

  - **Hashtable** which maps keys to values.
  - **StringTokenizer** which breaks a String down into individual words.
  - **Vector** which holds an array of objects whose size can change dynamically
  - **Stack** which holds a last-in, first-out stack of objects.

- **JDBC for SQL operations** The package `java.sql.*` contains the classes needed by Java objects to extract data from the database using SQL statements.

API classes outside the database

Unlike other Java classes, the Java API is not installed into the database. Rather, they are present as files on disk.

User-defined classes

User-defined classes are installed into a database using the INSTALL statement.

Once installed, they are available from other classes in the database and are available from SQL as user-defined data types.

For information on installing classes, see "Installing Java classes into a database" on page 474.

Table values

Instances of classes that have been saved as a value in a table can be retrieved like any other data in a table.

For example, the following SQL statements would successfully return a reference to an instance of a Java class called JClass. The statements assume both the variable and the column named JCol are of type JClass; and the correct instance, as specified by the WHERE clause of the query, had been saved to the table named T1.

```
CREATE VARIABLE var JClass;
SET var = (SELECT JCol
            FROM T1
            WHERE JCol.empName = 'John Smith');
```

Java syntax in SQL statements

The following table illustrates how some aspects of Java syntax are carried out when Java references are made from SQL.

Language feature	Java syntax	SQL syntax
"." dot operator	"." (dot)	>> or "." (dot)
"." dot identifier	"." (dot)	"."
String literal (quotes)	"*String*"	'*String*'
Import statement	import *java_package*.*;	Does not apply
System.out.println(str)	prints str to command line	prints str to server window

The use of all the examples above, plus additional notes on Java syntax usage, are explained in the following sections.

Identifying Java methods and fields

The dot in SQL

In SQL statements, the dot is used to identify columns of tables, as in the following query:

```
SELECT employee.emp_id
FROM employee
```

The dot is also used in qualified object names to indicate object ownership:

```
SELECT emp_id
FROM dba.employee
```

The dot in Java

In Java instructions, the dot is used as an **operator** to invoke the methods and fields of a Java class or object. It is also used as an **identifier** to identify class name hierarchies, as in the fully qualified class name `java.util.Hashtable`.

The dot character gets used for two distinct purposes.

♦ As an identifier, the dot identifies the word that follows it as a name. This helps locate the package, class or class method being referenced.

♦ As an operator, the dot causes a method to be invoked.

In the following Java code fragment, the dot is an identifier on the left hand side of the first line of code. On right hand side of the first line, and on the second line of code, it is an operator.

```
java.util.Random rnd = new java.util.Random();
int i = rnd.nextInt();
```

Invoking Java methods from SQL

To indicate methods and fields of Java objects in a SQL statement, you can use either the dot or the double right angle bracket (>>). The dot operator looks more like Java, but the double right angle bracket is less ambiguous: it cannot be confused with a database object.

To indicate class names, you must use the dot.

>> in SQL is not the same as >> in Java
The double right angle bracket operator is only used in SQL statements where a Java dot operator is otherwise expected. Within a Java class, the double right angle bracket is not a replacement for the dot operator and has a completely different meaning in its role as the right bit shift operator.

For example, the following batch of SQL statements is valid:

```
CREATE VARIABLE rnd java.util.Random;
SET rnd = NEW java.util.Random();
SELECT rnd>>nextInt();
```

The result of the SELECT statement is a randomly generated integer.

Using the variable created in the previous SQL code example, the following SQL statement illustrates the correct use of a class method.

```
SELECT java.lang.Math.abs( rnd>>nextInt() );
```

In the above statement, **abs** is a method, but it is a class method. It is only the name of the method at the end of the name of class. The dot preceding it is only used for identification, therefore it is not substituted with a double right angle bracket, as is the case for the **nextInt()** method.

The dot character is an operator when it is acting on an instance of a class.

For example the following code is valid, even though the ">>" substitute operator is being used with a static method.

```
CREATE VARIABLE mth java.lang.Math;
SET mth = NEW java.lang.Math();
```

```
SELECT mth>>abs( rnd>>nextInt() )
```

This is successful because the target of the operation is an object, not a class.

Java is case sensitive

Java syntax works as you would expect it to, and SQL syntax is unaltered by the presence of Java classes. This is true even if the same SQL statement contains both Java and SQL syntax. It's a simple statement but with far-reaching implications.

Java is case sensitive. The Java class **`FindOut`** is a completely different class from the class **`Findout`**. SQL is case insensitive in many ways: the case of SQL keywords and identifiers is ignored.

Java case sensitivity is preserved even when embedded in a SQL statement that is case insensitive. The Java parts of the statement must be case sensitive, even though the parts previous to and following the Java syntax can be written in upper or lower case.

For example the following SQL statements will successfully execute because the case of Java objects, classes and operators is respected, even though there is variation in the case of the remaining SQL parts of the statement.

```
SeLeCt java.lang.Math.random();
```

Data types

When a Java class is used as a data type for a column, it is being used as a user-defined SQL data type. However, it is still case sensitive. This convention prevents ambiguities with Java classes that differ only in case.

Strings in Java and SQL

String literals are identified in Java by a set of double quotes, as in the following Java code fragment.

```
String str = "This is a string";
```

In SQL, however, strings are marked by single quotes, and double quotes indicate an identifier, as illustrated by the following SQL statement.

```
INSERT INTO TABLE DBA.t1
VALUES( 'Hello' )
```

You should always use the double quote in Java source code, and single quotes in SQL statements.

For example, the following SQL statements are valid.

```
CREATE VARIABLE str char(20);
SET str = NEW java.lang.String( 'Brand new object' )
```

The following Java code fragment is also valid, if used within a Java class.

```
String str = new java.lang.String(
            "Brand new object" );
```

Printing to the command line

Printing to the command line is a quick method of checking variable values and execution results at various points of code execution. When the method in the second line of the following Java code fragment is encountered, the string argument it accepts is printed out to the command line.

```
String str = "Hello world";
System.out.println( str );
```

In Adaptive Server Anywhere, this method causes the string argument to be printed out to the server window.

Executing the above Java code within the database is the equivalent of the following SQL statement.

```
MESSAGE 'Hello world'
```

Using the main method

When a class contains a **`main`** method that matches the following method declaration, it is executed automatically by most Java run time environments, such as the Sun Java interpreter. Normally, this static method will execute when the class is loaded, without needing to be explicitly invoked.

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

It is a useful class for testing the functionality of Java objects: you are always guaranteed this method will be called first, when the Sun Java runtime system is started.

In Adaptive Server Anywhere the **`main`** method is not automatically invoked. In Adaptive Server Anywhere the Java runtime system is always available. The functionality of objects and methods can be tested in an ad hoc, dynamic manner using SQL statements. In many ways this is a far more flexible method of testing Java class functionality.

Scope and persistence

SQL variables are persistent only for the duration of connection. This is unchanged from previous versions of Adaptive Server Anywhere, and is unaffected whether the type of the variable is a Java class or a native SQL data type.

The persistence of Java classes is analogous to tables in a database: Tables exist in the database until they are dropped, regardless of whether they hold data or ever get used. Java classes that have been installed to a database are similar: they are available for use until they are explicitly removed.

All installed Java classes are available until they are removed from the database with a REMOVE statement.

For more information on removing classes, see "REMOVE statement" on page 530 of the book *Adaptive Server Anywhere Reference Manual*.

A class method in an installed Java class can be called at any time, from a SQL statement. The following statement can be executed anywhere SQL statements can be executed.

```
SELECT java.lang.Math.abs(-342)
```

A Java object is only available in two forms: as the value of a variable, or a value in a table. The scope of a specific instance of a Java class is limited to: whether or not it exists; and whether or not the field or method of the variable or table that contains the instance can be accessed (as determined by its access modifier: public, private or protected).

Java escape characters in SQL statements

In Java code, escape characters can be used to insert certain unique characters into strings when a string is being declared. Consider the following Java code, which inserts a new line and tab in front of a sentence containing an apostrophe.

```
String str = "\n\t\This is the object\'s string
literal.";
```

The use of Java escape characters is permitted in Adaptive Server Anywhere only when it is being used by Java classes.

From within SQL, however, the rules that apply to strings in SQL must be followed.

For example, to pass a string value to a field using a SQL statement, the following statement could be used, but the Java escape characters could not.

```
SET obj>>str = '\nThis is the object''s string field';
```

☞ for more information on SQL string handling rules, see "Statement elements" on page 180 of the book *Adaptive Server Anywhere Reference Manual*.

Keyword conflicts

SQL keywords can conflict with the names of Java classes, including API classes. This occurs when the name of a class, such as the Date class, which is a member of the **`java.util.*`** package, is referenced. SQL reserves the word **Date** for use as a keyword, even though it also the name of a Java class.

When such ambiguities are encountered double quotes can be used to identify that the word in question is not being used as the SQL reserved word.

For example, the following SQL statement causes an error because Date is a keyword and its use is reserved within SQL.

```
-- This statement is incorrect
CREATE VARIABLE dt java.util.Date
```

However the following two statements work correctly because the word Date is quoted.

```
CREATE VARIABLE dt java.util."Date";
SET dt = NEW java.util."Date"(1997, 11, 22, 16, 11, 01)
```

The variable **dt** now contains the date: November 22, 1997, 4:11 p.m.

Use of import statements

It is common in a Java class declaration to include an import statement to access other, external classes. This makes classes that are members of Java package specified by the import statement available to the class being declared using only abbreviated names. External classes can always be accessed using a fully qualified name.

For example, the Stack class of the **`java.util`** package can be reference in a class two ways:

♦ explicitly using the name **`java.util.Stack`**, or

♦ implicitly using the name **`Stack`** if class declaration contains the following import statement.

```
import java.util.*;
```

All classes used in Adaptive Server Anywhere are compiled, therefore the Java compiler has already checked if a reference to a class is properly qualified.

The class being used within Adaptive Server Anywhere may or may not have used an import statement, depending on the Java developer's design.

Classes further up in the hierarchy must also be installed.

The only restriction on compiled classes is that a class referenced by another class, either explicitly with a fully qualified name or implicitly using an import statement, must also be installed in the database.

The import statement works as intended within compiled classes, however, within the Adaptive Server Anywhere runtime environment, there is no equivalent to the import statement. All class names used in SQL statements or stored procedures must be fully qualified.

For example, to create a variable of type String, the class is referenced using the fully qualified name: **`java.lang.String`**.

Using the CLASSPATH variable

The CLASSPATH environment variable is used by Sun's Java runtime environment and by the Sun JDK Java compiler to locate the classes that are referenced within Java code. A CLASSPATH variable provides the link between Java code such as …

```
import java.io.*
```

… and the actual file path or URL location of the classes being referenced.

For example, the above statement allows all the classes in the **`java.io`** package to be referenced without a fully qualified name. Only the class name is required in the following Java code to use classes from the **`java.io`** package. The CLASSPATH environment variable on the system where the Java class declaration is to be compiled must include the location of the java directory, the root of the **`java.io`** package.

CLASSPATH ignored at runtime

The CLASSPATH environment variable does not affect the Adaptive Server Anywhere runtime environment for Java during the execution of Java operations.

When a class is installed to a database, its full package name is retained, and so the CLASSPATH environment variable is not used.

CLASSPATH used to install classes

The CLASSPATH variable can, however, be used to locate a file during the installation of classes. For example, the following statement installs a user-created Java class to a database, but only specifies the name of the file, not its full path and name. (Note that this statement involves no Java operations.)

```
INSTALL JAVA NEW
```

```
FROM FILE 'Invoice.class'
```

If the file specified is in a directory or zip file specified by the CLASSPATH environmental variable, Adaptive Server Anywhere will successfully locate the file and install the class.

Public fields

It is a common practice in object oriented programming to define class fields as private and make their values available only through public methods.

Many of the examples used in this documentation render fields as public so that examples are more compact and easier to read. It is also the case, however, that using public fields in Adaptive Server Anywhere does offer a performance advantage over accessing public methods. However, in order to update a field in a table, the class must have a method that sets the value of the field.

The general convention followed in this documentation is that a user-created Java class designed for use in Adaptive Server Anywhere exposes its main values in its fields. Methods are used to contain computational automation and logic that may act on these fields.

This applies to class-level variables only. Local variables, those declared within methods only, do not have the public or private access modifiers.

A Java in the database exercise

Invoking Java operations from within a SQL environment will likely be new to Java developers and database users alike.

This section is a primer for invoking Java operations on Java classes and objects using SQL statements. The examples use the Invoice class created in "A Java seminar" on page 439.

Case sensitivity
Java is case sensitive, so the portions of the following examples in this section pertaining to Java syntax are written using the correct case. SQL syntax is rendered in upper case.

A sample Java class

The following class declaration is used throughout all the following examples.

Compiled code available
Source code and compiled versions of all Java classes outlined in the documentation are included with Adaptive Server Anywhere. The file *Invoice.java*, is available and can be compiled and installed into a database.

```
public class Invoice {

  // Fields
  public String lineItem1Description;
  public double lineItem1Cost;

    public String lineItem2Description;
    public double lineItem2Cost;

    // An instance method
    public double totalSum() {
        double runningsum;
        double taxfactor = 1 + Invoice.rateOfTaxation();

        runningsum = lineItem1Cost + lineItem2Cost;
        runningsum = runningsum * taxfactor;

        return runningsum;
    }
    // A class method
    public static double rateOfTaxation() {
        double rate;
        rate = .15;

        return rate;
    }
}
```

> ***Caution: use a Java-enabled database***
> *The following section assumes the database you are connected to is Java-enabled. For more information, see "Java-enabling a database" on page 469.*

Installing Java classes

Any Java class must be installed to a database before it can be used. You can install classes from Sybase Central or Interactive SQL.

❖ **To install the Invoice class to the sample database from Sybase Central:**

1 From Sybase Central, choose Tools➤Connect➤Adaptive Server Anywhere.

2 Select the ASA 6.0 Sample data source, and connect.

3 Open the Java Objects folder and double click Add Java Class or Jar. The Install a New Java Object wizard is displayed.

4 Select the Java Class File option, and click Next.

5 Use the Browse button to locate *Invoice.class*, which is in the *jxmp* subdirectory of your Adaptive Server Anywhere installation directory.

❖ **To install the Invoice class to the sample database from Interactive SQL:**

♦ Start Interactive SQL and connect to the ASA 6.0 Sample ODBC data source. Enter the following SQL statement:

```
INSTALL JAVA NEW
FROM FILE 'path\jxmp\Invoice.class';
```

where *path* is your Adaptive Server Anywhere installation directory.

Notes

- At this point no Java operations have taken place. The class has been installed into the database and is ready to be used as the data type of a variable or column in a table.
- Changes made to the class file from now on are *not* automatically reflected in the copy of the class in the database.

For more information on installing classes, and for information on updating an installed class, see "Installing Java classes into a database" on page 474.

Creating SQL variables of Java class type

The following statement creates a SQL variable named **Inv** of type **Invoice**, where **Invoice** is the Java class that you installed to a database.

```
CREATE VARIABLE Inv Invoice;
```

Once any variable has been created, it can only be passed as a value that is the same data type as its declared data type. In this case, the variable **Inv** can only contain a reference to an object of type **Invoice**.

Perform the following statement to identify the current value of the variable **Inv**.

```
SELECT IFNULL(Inv, 'No object referenced',
  'Variable not null: contains object reference')
```

The SQL **IFNULL** function accepts three arguments: the first is the variable to be tested; the second is the expression selected if the value is NULL; the third is the expression selected if the variable is not NULL.

At this point the variable **Inv** contains a NULL because no value has been passed to it. The only value it could contain is an object reference where the object is of type **Invoice**.

To pass a value to **Inv**, the default constructor of the **Invoice** class needs to be invoked. The **NEW** keyword is used to indicate a constructor is being invoked and an object reference is being returned.

```
SET Inv = NEW Invoice();
```

The **Inv** variable now has a reference to a Java object. To verify this a number of select statements can be executed using the variable.

For example the following statement shows the variable contains a value.

```
SELECT IFNULL(Inv, 'No object referenced',
    'Variable not null: contains object reference')
```

The **Inv** variable should contain a reference to a Java object of type **Invoice**. Using this reference, any of the object's fields can be accessed and its methods can be invoked.

Invoking Java operations

If a variable (or column value in a table) contains a reference to a Java object, then the fields of the object can be passed values and its methods can be invoked.

For example, a variable of type Invoice (a user-created class) that contains a reference to an Invoice object will have four fields, the value of which can be set using SQL statements.

Passing values to fields

The following SQL statements set the field values for just such a variable.

```
SET Inv>>lineItem1Description = 'Work boots';
SET Inv>>lineItem1Cost = '79.99';
SET Inv>>lineItem2Description = 'Hay fork';
SET Inv>>lineItem2Cost = '37.49';
```

Each line in the SQL statements above passes a value to a field in the Java object referenced by **Inv**. This can be shown by performing a select statement against the variable. Any of the following SQL statements return the current value of a field in the Java object referenced by **Inv**.

```
SELECT Inv>>lineItem1Description;
SELECT Inv>>lineItem1Cost;
SELECT Inv>>lineItem2Description;
SELECT Inv>>lineItem2Cost;
```

Each line of the above lines can now be used as an expression in other SQL statements. For example the following SQL statement can be executed if you are currently connected to the sample database, *asademo.db*, and have executed the above SQL statements.

```
SELECT * FROM PRODUCT
   WHERE unit_price < Inv>>lineItem2Cost;
```

Invoking methods

The **Invoice** class has one instance method, which can be invoked when an object of type **Invoice** has been created.

The following SQL statement invokes the **totalSum()** method of the object referenced by the variable **Inv**. It returns the sum of the two cost fields plus the tax charged on this sum.

```
SELECT Inv>>totalSum();
```

Calling methods versus referencing fields

Notice the round brackets following the name of the method used in the above SQL statement.

A key difference between the object's fields and its methods is that methods are invoked which causes them to perform an action and return a value (even if the value is void). Fields are referenced in order to access the values they may contain.

The **totalSum()** method takes no arguments but returns a value. The brackets are used even though the method takes no arguments because a Java operation is being invoked.

As indicated by the Invoice class definition outlined at the beginning of this section, the `totalSum` instance method makes use of the class method `rateOfTaxation`.

This class method can be accessed directly from a SQL statement.

```
SELECT Invoice.rateOfTaxation();
```

Notice the name of the class is being used, not the name of a variable containing a reference to an `Invoice` object. This is consistent with the way Java handles class methods, even though it is being used in a SQL statement. A class method can be invoked even if no object based on that class has been instantiated.

Class methods do not require an instance of the class in order to work properly, but they can still be invoked on an object. The following SQL statement yields the same results as the previously executed SQL statement.

```
SELECT Inv>>rateOfTaxation();
```

Saving Java objects in tables

When a class is installed in a database, it is available as a new data type. Columns in a table can be of type *Javaclass* where *Javaclass* is the name of an installed Java class.

For example, using the **Invoice** class that was installed at the beginning of this section, the following SQL statement can be executed.

```
CREATE TABLE T1 (
    ID int,
    JCol Invoice
);
```

The column named **JCol** only accepts objects of type **Invoice**, which is an installed Java class. This means only objects can be passed in as values in the JCol column.

There are at least two methods for creating a Java object and adding it to a table as the value of a column. The first method, creating a variable, was outlined in a previous section "Creating SQL variables of Java class type" on page 460.

Assuming the variable **Inv** contains a reference to a Java object of type **Invoice**, the following SQL statement will add a row to the table **T1**.

```
INSERT INTO T1
VALUES( 1, Inv );
```

An object has been added to the table **T1**. Select statements can be issued involving the fields and methods of the objects in the table.

For example the following SQL statement will return the value of the field lineItem1Description for all the objects in the table T1 (right now, there should only be one object in the table).

```
SELECT ID, JCol>>lineItem1Description
FROM T1;
```

Similar select statements involving other fields and methods of the object can be executed.

A second method for creating a Java object and adding it to a table involves the following expression, which always creates a Java object and returns a reference to it:

```
NEW Javaclassname()
```

This expression can be used in a number of ways. For example, the following SQL statement creates a Java object and inserts it into the table **T1**.

```
INSERT INTO T1
VALUES ( 2, NEW Invoice() );
```

The following SQL statement verifies that these two objects have been saved as values of column **JCol** in the table **T1**.

```
SELECT ID, JCol>>totalSum()
FROM t1
```

The results of the **JCol** column (the second row returned by the above statement) should be 0, because the fields in that object have no values and the **`totalSum`** method is a calculation of those fields.

Returning an object using a query

An object can also be retrieved from a table that has a Java class as the type of one of its columns. In the following series of statements a new variable is created and passed a value (it can only contain an object reference where the object is of type Invoice). The object reference passed to the variable was generated using the table T1.

```
CREATE VARIABLE Inv2 Invoice;
SET Inv2 = (select JCol from T1 where ID = 2);
SELECT IFNULL(Inv2, 'No object referenced',
   'Variable not null: contains object reference');
SET Inv2>>lineItem1Description = 'Sweet feed';
SET Inv2>>lineItem2Description = 'Drive belt';
```

Take note that the value for the **`lineItem1Description`** field and **`lineItem2Description`** have been changed in the variable **Inv2** but not in the table that was the source for the value of this variable.

This is consistent with the way SQL variables are currently handled: the variable **Inv** contains a reference to a Java object. The value in the table that was the source of the variable's reference is not altered until an UPDATE SQL statement is executed.