sun microsystems

JavaSoft

JDBC ™: A Java SQL API

This is a draft specification for the JDBC™ API, which is a Java™ application programming interface to SQL databases.

This draft (version 0.50) is the first general public distribution of the JDBC specification. We will be taking public comments on the spec over the next 90 days, and will release a final version in June.

Please send reviewer comments to the authors at:

jdbc@wombat.eng.sun.com

Because of the volume of interest in JDBC, we will not be able to respond individually to comments or questions. However, we greatly appreciate your feedback, and we will read and carefully consider all mail that we receive.

For marketing and business questions please contact Jim Herriot at (408) 343 1429 (herriot@eng.sun.com)

Version 0.50

Graham Hamilton & Rick Cattell

March 5, 1996

Contents

1 Introduction

Many vendors and customers are now looking for easy database access for Java applications. Since Java is robust, secure, easy to use, easy to understand, and automatically downloadable on a network, it is an excellent language basis for the development of database applications. It offers many advantages over C, C++, Smalltalk, BASIC, COBOL, and 4GLs for a wide variety of uses. A number of companies are already working on Java connectivity to DBMSs.

Many Java application developers would like to write code that is independent of the particular DBMS or database connectivity mechanism being used, and we believe that a DBMS-independent interface is also the fastest way to implement access to the wide variety of DBMSs. So, we think it would be useful to the Java community to define a generic SQL database access framework which provides a uniform interface on top of a variety of different database connectivity modules. This allows programmers to write to a single database interface, enables DBMS-independent Java application development tools and products, and allows database connectivity vendors to provide a variety of different connectivity solutions.

We see our immediate priority is to define a common low-level Java DataBase Connectivity (JDBC) API that supports basic SQL functionality. This will then allow the development of higher-level DB access tools and APIs.

Fortunately we don't need to design a SQL API from scratch. We can base our work on the X/Open SQL CLI (Call Level Interface) which is also the basis for Microsoft's ODBC interface. We see our main task as defining a natural Java interface to the basic abstractions and concepts defined in the X/Open CLI.

It is important that the JDBC API be accepted by database vendors, connectivity vendors, ISVs, and application writers. We believe that basing our work on the ODBC abstractions is likely to make this acceptance easier, and technically ODBC seems a good basis for our design.

ODBC is not appropriate for *direct* use from Java, since it is a C interface; calls from Java to native C code have a number of drawbacks in the security, implementation, robustness, and automatic portability of applications. Thus, we have constructed an API that can easily be implemented on top of ODBC short-term, and that can be implemented in other ways longer term.

1.1 Acknowledgments

We would like to thanks the many early reviewers in the database, database connectivity, and database tools communities who provided comments and advice on earlier drafts of JDBC. Their input has improved the specification immeasurably. The errors and omissions that remain, are however, all our own work.

2 Goals and philosophy

This section outlines the main goals and philosophy driving the API design.

2.1 A SQL level API

Our main goal is to define a "call-level" SQL interface for Java. This means the focus is on executing raw SQL statements and retrieving their results. We expect that higher-level APIs will be defined as well, and these will probably implemented on top of this base level. Examples of higher-level APIs are direct transparent mapping of tables to Java classes, semantic tree representations of more general queries, and an embedded SQL syntax for Java.

We expect that various application builder tools will emit code that uses our API. However we also intend that the API be usable by human programmers, especially because there is no other solution available for Java right now.

2.2 SQL Conformance

Database systems support a wide range of SQL syntax and semantics, and they are not consistent with each other on more advanced functionality such as outer joins and stored procedures. Hopefully with time the portion of SQL that is truly standard will expand to include more and more functionality. In the meantime, we take the following position:

- JDBC allows any query string to be passed through to an underlying DBMS driver, so an application may use as much SQL functionality as desired at the risk of receiving an error on some DBMSs. In fact, an application query need not even be SQL, or it may be a specialized derivative of SQL, e.g. for document or image queries, designed for specific DBMSs.
- In order to pass JDBC compliance tests and to be called "JDBC COMPLIANT™" we require that a driver support at least ANSI SQL-2 Entry Level. This gives applications that want wide portability a guaranteed least common denominator. We believe ANSI SQL-2 Entry Level is reasonably powerful and is reasonably widely supported today.

2.3 JDBC must be implementable on top of common database interfaces

We need to ensure that the JDBC SQL API can be implemented on top of common SQL level APIs, in particular ODBC. This requirement has colored some parts of the specification, notably the handling of OUT and INOUT parameters.

2.4 Provide a Java interface that is consistent with the rest of the Java system

There has been a very strong positive response to Java. To a large extent this seems to be because the language and the standard runtimes are perceived as being consistent, simple, and powerful.

As far as we can, we would like to provide a Java database interface that builds on and reinforces the style and virtues of the existing core Java classes.

2.5 Keep it simple

We would prefer to keep this base API as simple as possible, at least initially. In general we would prefer to provide a single mechanism for performing a particular task, and avoid providing duplicate mechanisms. We will extend the API later if any important functionality is missing.

2.6 Use strong, static typing wherever possible

We would prefer that the JDBC API be strongly typed, with as much type information as possible expressed statically. This allows for more error checking at compile time.[1]

Because SQL is intrinsically dynamically typed, we may encounter type mismatch errors at run-time where for example a programmer expected a SELECT to return an integer result but the database returned a string "foo". However we would still prefer to allow programmers to express their type expectations at compile time, so that we can statically check as much as possible. We will also support dynamically typed interfaces where necessary (see particularly Chapter 14).

2.7 Keep the common cases simple

We would like to make sure that the common cases are simple, and that the uncommon cases are doable.

A common case is a programmer executing a simple SQL statement (such as a SELECT, INSERT, UPDATE or DELETE) without parameters, and then (in the case of SELECT statement) processing rows of simple result types. A SQL statement with IN parameters is also common.

Somewhat less common, but still important, is when a programmer invokes a SQL statement using INOUT or OUT parameters. We also need to support SQL statements that read or write multi-megabyte objects, and less common cases such as multiple result sets returned by a SQL statement.

We expect that metadata access (e.g. to discover result-set types, or to enumerate the procedures in a database) is comparatively rare and is mostly used by sophisticated programmers or by builder tools.

2.8 Use multiple methods to express multiple functionality

One style of interface design is to use a very small number of procedures and offer a large number of control flags as arguments to these procedures, so that they can be used to effect a wide range of different behavior.[2]

In general the philosophy of the Java core classes has been to use different methods to express different functionality. This tends to lead to a larger number of methods, but makes each meth-

1. This contrasts with ODBC, which makes heavy use of "void *". The Java equivalent of "void *" would be the use of "Object". We've tried to resist this temptation.
2. For example, the ODBC SQLBindParameter procedure can be used to specify a wide range of behavior, from simple variable binding, to delivering data at statement execution time.

od easier to understand. This approach has the major advantage that programmers who are learning how to use the basic interface aren't confused by having to specify arguments related to more complex behavior.

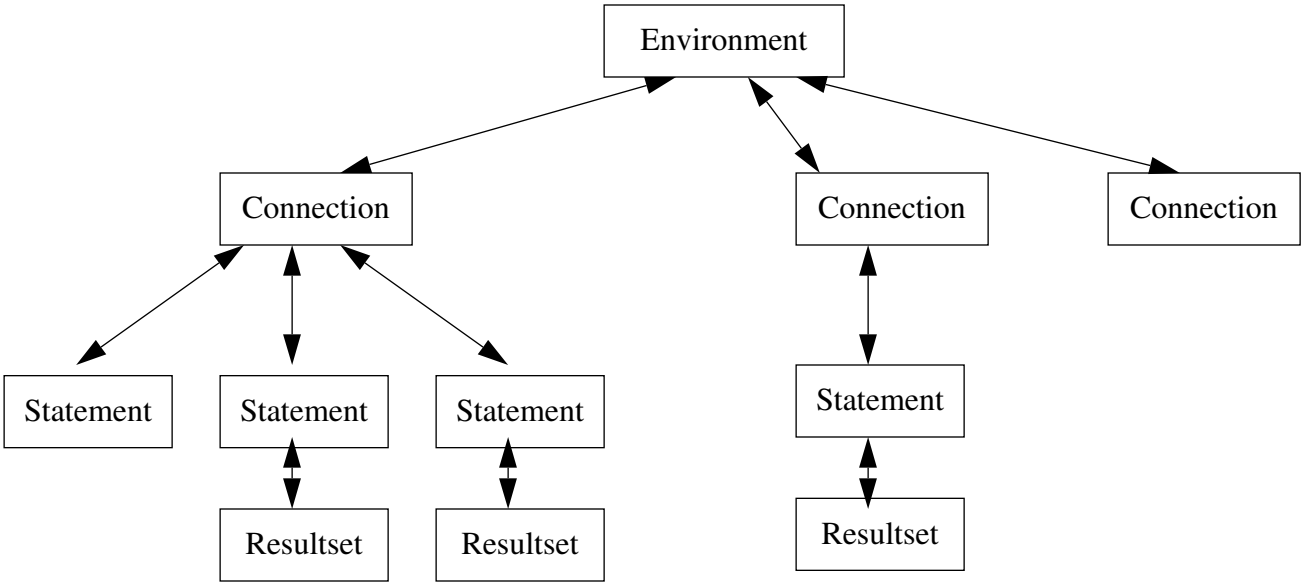
We've tried to adopt the same approach for the JDBC interface, and in general have preferred to use multiple methods rather than using multi-purpose methods with flag arguments.

3 Overview of the major interfaces

There are two major sets of interfaces. First there is a JDBC API for application writers. Second there is a lower level JDBC Driver API.

3.1 The JDBC API

The JDBC API is expressed as a series of abstract Java interfaces that allow an application programmer to open connections to particular databases, execute SQL statements, and process the results.

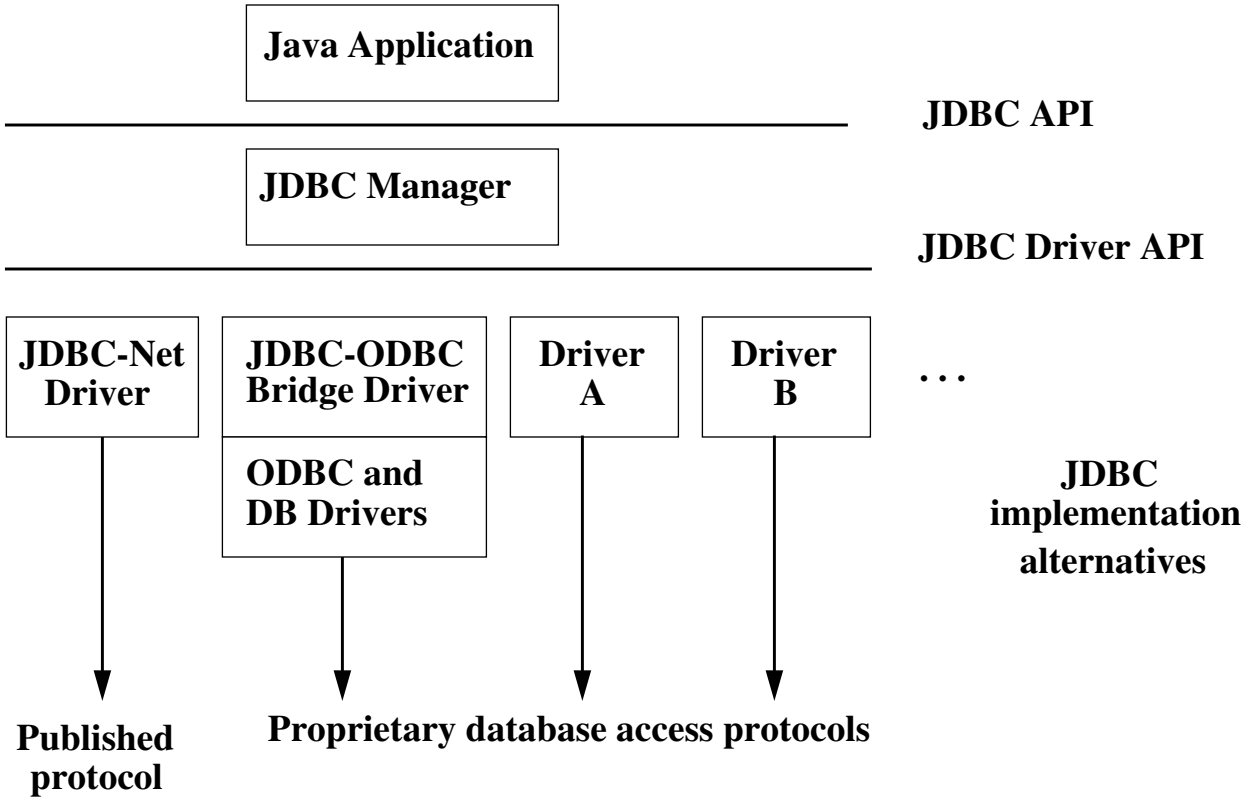


The most important interfaces are:

- java.sql.Environment which provides support for creating new database connections
- java.sql.Connection which represents a connection to a particular database
- java.sql.Statement which acts as a container for executing a SQL statement on a given connection
- java.sql.ResultSet which controls access to the row results of a given Statement

In addition the java.sql.Statement interface has two important sub-types: java.sql.PreparedStatement for executing a pre-compiled SQL statement, and java.sql.CallableStatement for executing a call to a database stored procedure.

The following chapters provide more information on how these interfaces work, and the complete Java definitions are given in Chapter 13.



3.2 The JDBC Driver Interface

The java.sql.Driver interface is fully defined in Chapter 9. For the most part the database drivers simply need to provide implementations of the abstract classes provided by the JDBC API. Specifically, each driver must provide implementations of java.sql.Connection, java.sql.Statement, java.sql.PreparedStatement, java.sql.CallableStatement and java.sql.ResultSet.

In addition, each database driver needs to provide a class which implements the java.sql.Driver interface used by the generic java.sql.Environment class when it needs to locate a driver for a particular database URL.

An obvious driver to provide would be an implementation of JDBC on top of ODBC, as shown in JDBC-ODBC bridge in the picture. Since JDBC is patterned after ODBC, this implementation is small and efficient.

Another useful driver would be one that goes directly to a DBMS-independent network protocol. It would be desirable to publish the protocol to allow multiple server implementations, e.g. on top of ODBC or on specific DBMSs (although there are already products that use a fixed protocol such as this, we are not yet trying to standardize it). Only a few optimizations are needed on the client side, e.g. for schema caching and tuple look-ahead, and the JDBC Manager it-

self is very small and efficient as well.[1] The net result is a very small and fast all-Java client side implementation that speaks to any server speaking the published protocol.

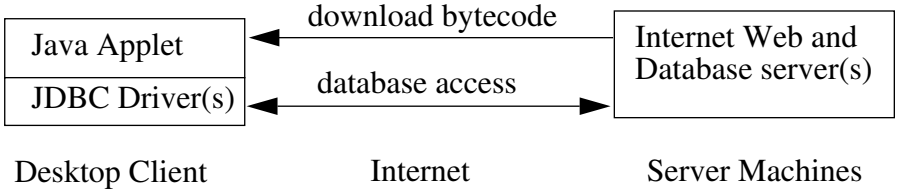
1. Most JDBC calls go directly from the application to the driver, so the driver manager contains very few methods; in contrast each ODBC call goes through a stub in the ODBC driver manager.

4 Scenarios for use

Before looking at specifics of the JDBC API, an understanding of typical use scenarios is helpful. There are two common scenarios that must be treated differently for our purposes: *applets* and *applications*.

4.1 Applets

The most publicized use of Java to date is for implementing *applets* that are downloaded over the net as parts of web documents. Among these will be database access applets, and these applets could use JDBC to get to databases.



For example, a user might download a Java applet that displays historical price graphs for a custom portfolio of stocks. This applet might access a relational database over the Internet to obtain the historical stock prices.

The most common use of applets may be across untrusted boundaries, e.g. fetching applets from another company on the Internet. Thus, this scenario might be called the “Internet” scenario. However, applets might also be downloaded on a local network where client machine security is still an issue.

Typical applets differ from traditional database applications in a number of ways:

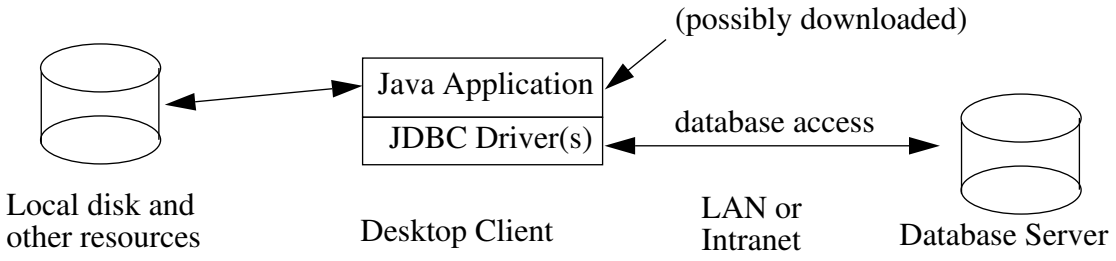
- Untrusted applets are severely constrained in the operations they are allowed to perform. In particular, they are not allowed any access to local files and they are not allowed to open network connections to arbitrary hosts.
- Applets on the Internet present new problems with respect to identifying and connecting to databases.[1]
- Performance considerations for a database connectivity implementation differ when the database may be halfway around the world. Database applets on the Internet will experience quite different network response times than database applications on a local area network.

4.2 Applications

Java can also be used to build normal applications that run like any shrink-wrapped or custom application on a client machine. We believe this use of Java will become increasingly common as better tools become available for Java and as people recognize the improved programming productivity and other advantages of Java for application development. In this case the Java

1. For example, you could not depend on your database location or driver being in a .INI file or local registry on the client’s machine, as in ODBC.

code is trusted and is allowed to read and write files, open network connections, etc., just like any other application code.



Perhaps the most common use of these Java applications will be within a company or on an "Intranet," so this might be called the Intranet scenario. For example, a company might implement all of its corporate applications in Java using GUI building tools that generate Java code for forms based on corporate data schemas. These applications would access corporate database servers on a local or wide area network. However, Java applications could also access databases through the Internet.

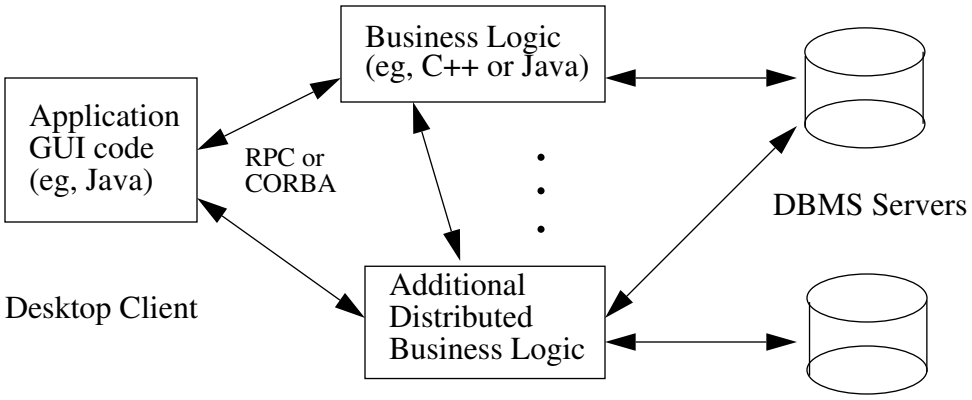
The Java application and "Intranet" cases differ from applets in a number of ways. For example, the most natural way to identify a database is typically for the user or application to specify a database name, e.g. "Customers" or "Personnel". The users will expect the system to locate the specific machine, DBMS, JDBC driver, and database.

4.3 Other scenarios

There are some other special cases of interest:

- *Trusted applets* are applets that have convinced the Java virtual machine that they can be trusted. They might be trusted because they have been signed with a particular cryptographic key, or because the user has decided to trust applets from a particular source. We will treat these applets the same as applications for security purposes, but they may behave more like applets for other purposes, e.g. locating a database on the Internet.
- *Three-tier access* to databases may be used, in contrast to direct client/server access from Java GUIs to DBMS servers. In this scenario, Java applications make calls to a "middle tier" of services on the net whose implementations in turn access databases. These calls might be made through RPC (remote procedure call) or through an ORB (object request broker), and in either case the middle tier might best be defined using an object paradigm, e.g. "customer objects" with operations for customer invoicing, address changes, and other transactions.

We expect that three-tier access will become more common because it is attractive to the MIS director to explicitly define the legal operations on their corporate data rather than allowing direct unrestricted updates to the DBMS servers. Also, the three-tier architecture can provide performance advantages in many cases.



Today, the middle tier is typically implemented in a language such as C or C++. With the introduction of optimizing compilers translating Java byte codes into efficient machine-specific code, the middle tier may practically be implemented in Java; Java has many valuable qualities (robustness, security, multi-threading) for these purposes. JDBC will be of use for this middle tier.

5 Security considerations

Based on the previous discussion, there are two main JDBC scenarios to consider for security purposes:

- In the case of Java applications, the Java code is "trusted". We also consider trusted applets in this category for security purposes.
- In contrast, untrusted Java applets are not permitted access to local files and or network connections to arbitrary hosts.

5.1 JDBC and untrusted applets

JDBC should follow the standard applet security model. Specifically:

- JDBC should assume that normal unsigned applets are untrustworthy
- JDBC should not allow untrusted applets access to local database data
- If a downloaded JDBC Driver registers itself with the JDBC Environment, then JDBC should only use that driver to satisfy connection requests from code which has been loaded from the same source as the driver.
- An untrusted applet will normally only be allowed to open a database connection back to the server from which it was downloaded
- JDBC should avoid making any automatic or implicit use of local credentials when making connections to remote database servers.

If the JDBC Driver level is completely confident that opening a network connection to a database server will imply no authentication or power beyond that which would be obtainable by any random program running on any random internet host, then it may allow an applet to open such a connection. This will be fairly rare, and would require for example, that the database server doesn't use IP addresses as a way of restricting access.

These restrictions for untrusted applets are fairly onerous. But they are consistent with the general applet security model and we can see no good way of relaxing them.

5.2 JDBC and Java applications

For a normal Java application (i.e. all Java code other than untrusted applets) JDBC should happily load drivers from the local classpath and allow the application free access to files, remote servers, etc.

However as with applets, if for some reason an untrusted sun.sql.Driver class is loaded from a remote source, then that Driver should only be used with code loaded from that same source.

5.3 Implementing security

When JDBC and the JDBC Drivers are loaded as part of an applet, then they will already be constrained by the applet security policy and will be unable to grant access to local files or make connections to random servers, even if they wished.

However, it is likely that the JDBC management layer and possibly some JDBC Drivers will be installed on the local disk. Some Drivers may use native methods to bridge to database li-

braries. In this case it is the responsibility of the Driver to ensure that it makes the appropriate checks when an applet tries to open a database connection.

Unfortunately, lower level database APIs (such as ODBC) aren't terribly helpful in exposing the security consequences of opening particular database connections. Each JDBC driver must make "worst case" assumptions and deny access unless it is confident that the connection grants no power to a malicious applet.

6 Database connections

(For the full interface descriptions see the Java interfaces in Chapter 13.)

6.1 Opening a connection

When you want to access a database you may obtain a java.sql.Connection object from the JDBC management layer's java.sql.Environment.getConnection method.

The Environment.getConnection method takes a URL string as an argument. The JDBC management layer will attempt to locate a driver that can connect to the database represented by the URL. The JDBC management layer does this by asking each driver in turn (see Section 6.2 below) to try to connect to the given URL. Driver's should examine the URL to see if it specifies a sub-protocol that they support (see Section 6.3 below) and if so they should attempt to connect to the specified database. If they succeed in establishing a connection then they should return an appropriate java.sql.connection object.

From the java.sql.Connection object it is possible to obtain java.sql.Statement and java.sql.PreparedStatement objects that can be used to execute SQL statements.

We also permit applications to bypass the JDBC management layer during connection open and explicitly select and use a particular driver.

6.2 Choosing between drivers

It may sometimes be the case that several JDBC drivers are capable of connecting to a given URL. For example, when connecting to a given remote database it might be possible to use either a JDBC-ODBC bridge driver, or a JDBC to generic network protocol driver, or to use a driver supplied by the database vendor.

JDBC allows users to specify a driver list by setting a Java property "sql.drivers". If this property is defined then it should be a space separated list of driver class names, such as "acme.wonder.Driver foobaz.openNet.Driver vendor.OurDriver".

When searching for a driver JDBC will use the first driver it finds that can successfully connect to the given URL. It will first try to use each of the drivers specified in the sql.drivers list, in the order given. It will then proceed to try to use each loaded driver in the order in which the drivers were loaded. It will skip any drivers which are untrusted code, unless they have been loaded from the same source as the code that is trying to open the connection (see the security discussion in Section 5).

6.3 URLs

6.3.1 Goals for JDBC database naming

We need to provide a way of naming databases so that application writers can specify which database they wish to connect to.

We would like this JDBC naming mechanism to have the following properties:

1. Different drivers can use different schemes for naming databases. For example, a JDBC-ODBC bridge driver may support simple ODBC style data source names, whereas a network protocol driver may need to know additional information so it can discover which hostname and port to connect to.
2. If a user downloads an applet that wants to talk to a given database then we would like to be able to open a database connection without requiring the user to do any system administration chores. Thus for example, we want to avoid requiring an analogue of the human-administered ODBC data source tables on the client machines. This implies that it should be possible to encode any necessary connection information in the JDBC name.
3. We would like to allow a level of indirection in the JDBC name, so that the initial name may be resolved via some network naming system in order to locate the database. This will allow system administrators to avoid specifying particular hosts as part of the JDBC name. However, since there are a number of different network name services (such as NIS, DCE, etc.) we do not wish to mandate that any particular network nameserver is used.

6.3.2 URL syntax

Fortunately the World Wide Web has already standardized on a naming system that supports all of these properties. This is the Uniform Resource Locator (URL) mechanism. So we propose to use URLs for JDBC naming, and merely recommend some conventions for structuring JDBC URLs.

We recommend that JDBC URL's be structured as:

jdbc:<subprotocol>:<subname>

Where a subprotocol names a particular kind of database connectivity mechanism that may be supported by one or more drivers. The contents and syntax of the subname will depend on the subprotocol.

If you are specifying a network address as part of your subname, we recommend following the standard URL naming convention of "//hostname:port/subsubname" for the subname. The subsubname can have arbitrary internal syntax.

6.3.3 Example URLs

For example, in order to access a database through a JDBC-ODBC bridge, one might use a URL like:

jdbc:odbc:fred

In this example the subprotocol is "odbc" and the subname is a local ODBC data source name "fred". A JDBC-ODBC driver can check for URLs that have subprotocol "odbc" and then use the subname in an ODBC SQLConnect.

If your are using some generic database connectivity protocol "dbnet" to talk to a database listener you might have a URL like:

jdbc:dbnet://wombat:356/fred

In this example the URL specifies that we should use the "dbnet" protocol to connect to port 356 on host wombat and then present the subsubname "fred" to that port to locate the final database.

If you wish to use some network name service to provide a level of indirection in database names, then we recommend using the name of the naming service as the subprotocol. So for example one might have a URL like:

jdbc:dcenaming:accounts-payable

In this example, the URL specifies that we should use the local DCE naming service to resolve the database name "accounts-payable" into a more specific name that can be used to connect to the real database. In some situations, it might be appropriate to provide a pseudo driver that performed a name lookup via a network name server and then used the resulting information to locate the real driver and do the real connection open.

6.3.4 Drivers can chose a syntax, and ignore other URLs.

In summary, the JDBC URL mechanism is intended to provide a framework so that different drivers can use different naming systems that are appropriate to their needs. Each driver need only understand a single URL naming syntax, and can happily reject any other URLs that it encounters.

6.3.5 Registering sub-protocol names

JavaSoft will act as an informal registry for JDBC sub-protocol names. Send mail to jdbc@wombat.eng.sun.com to reserve a sub-protocol name.

6.4 Connection arguments

When opening a connection you can pass in a java.util.Properties object. This object is a property set that maps between tag strings and value strings. Two conventional properties are "user" and "password". Particular drivers may specify and use other properties.

In order to allow applets to access databases in a generic way, we recommend that as much connection information as possible is encoded as part of the URL and that driver writers minimize their use of property sets.

6.5 Multiple connections

A single application can maintain multiple database connections to one or more databases, using one or more drivers.

6.6 Registering drivers

The JDBC management layer needs to know which database drivers are available. We provide two ways of doing this.

First, when the JDBC java.sql.Environment class initializes it will look for a "sql.drivers" property in the system properties. If the property exists it should consist of a space-separated list of driver class names. Each of the named classes should implement the java.sql.Driver interface.

The Environment class will attempt to load each named Driver class and the Driver class should then register itself with the Environment, using the Environment.registerDriver method.

Second, an application or applet can attempt to load additional drivers using the java.sql.Environment.loadDriver methods.

For security reasons the JDBC management layer will keep track of which class loader provided which driver and when opening connections it will only use drivers that come from the local filesystem or from the same classloader as the code issuing the getConnection request.

7 Passing parameters and receiving results

(For the full interface descriptions see the Java interfaces in Chapter 13.)

(See also the rejected "Holder" mechanism described in Appendix A.)

7.1 How parameters and results work in ODBC

Some background on how ODBC handles parameters and results may be useful here to establish some context.

ODBC provides two mechanisms for setting parameters and four mechanisms for receiving query results.

To set parameters you can:

(1) use SQLBindParameter to give ODBC a "void *" pointer to a statement parameter. When the statement executes ODBC will reach through this pointer to read IN or INOUT parameters and to write OUT or INOUT parameters.

(2) use SQLBindParameter to tell ODBC that the parameter data will be provided when the statement is executed. At statement execute time there is a rather complex little dance of having the execute return an interim result than means you should query for which parameter needs to be supplied and then use SQLPutData to supply that parameter. SQLputdata allows you to deliver large data items as a series of chunks, which makes it suitable for sending extremely large data.

To receive results from a query you can chose one of:

(1) Use SQLBindColumn to bind (using a "void *") pointers to variables that will be used to hold the values in a row. Then you can use SQLFetch to step through the rows of the results. As you step into each row, your bound variables will get updated with the contents of the row.

(2) You can use SQLFetch to step through the result rows and then use SQLGetData to retrieve part or all of a data value in the current row. SQLGetData takes a "void *" as the address where it should put the result. SQLGetData is particularly useful for retrieving very large LONGVARBINARY values, as it lets you read large values in fixed size chunks.

(3) You can use SQLBindColumn to do so-called "row-wise" binding where you allocate an arrays of structs representing result rows. You can then use SQLExtendedFetch to fetch multiple rows of results at a time directly into the array of result structs. This works in ODBC because they are using "void *" pointers to areas whose layout is fixed by the programmer. Unfortunately it's not clear how to do this in Java.

(4) You can use SQLBindColumn to do so called "column-wise" binding where you bind an array of variables to each column you're interested in and then use SQLExtendedFetch to fetch a set of rows all at once.

7.2 Query results

The result of executing a query Statement is a set of rows that are accessible via a java.sql.ResultSet object. The ResultSet object provides a set of "get" methods that allow access to the various columns of the current row. The ResultSet.next method can be used to move between the rows of the ResultSet.

```
// We're going to execute a SQL statement that will return a
// collection of rows, with column 1 as an int, column 2 as
// a String, and column 3 as an array of bytes.
java.sql.Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (r.next()) {
       // print the values for the current row.
       int i = r.getInteger(1);
       String s = r.getVarChar(2);
       byte b[] = r.getVarBinary(3);
       System.out.println("ROW = " + i + " " + s + " " + b[0]);
}
```

The various "get" methods take a ResultSet column number as their argument. Within a given row columns may be accessed in random order.

In an earlier version of the JDBC spec the various "get" methods took no arguments, but merely returned the next column value in left-to-right order. We (re)introduced a column number argument for two reasons. First, we were unsatisfied with the readibility of the resulting example code. We frequently found ourselves having to count through the various "get" calls in order to match them up with the columns specified in the SELECT statement. Second when we introduced the ResultSet.isNull() method we found that it was easier to write (and read) null checks when both isNull and the "get" methods took column numbers.

7.2.1 Null result values

To determine if a given result field is SQL "NULL" you can use the ResultSet.isNull method.

If you attempt to retrieve a SQL "NULL" value using one of the "get" methods, you will receive:

- A Java "null" value for those "get" methods that return Java objects.
- A zero value for getByte, getShort, getInt, getLong, getFloat, and getDouble.
- A false value for getBoolean.

In addition, a SQLWarning object of type java.sql.NullData will be added to the current Statement's list of warnings wherever a NULL value is read.

7.2.2 Retrieving very large row values.

JDBC allows arbitrarily large data to be retrieved using getLongaVarBinary and getLongVarChar, up to the limits imposed by the Statement.getMaxFieldSize value. However, application programmers may often find it convenient to retrieve very large data in smaller fixed size chunks.

To accommodate this, the ResultSet class can return java.io.Input streams from which data can be read in chunks. However each of these streams must be accessed immediately as they will be automatically closed on the next "get" call on the ResultSet. **This behavior reflects underlying implementation constraints on large blob access.**

For example:

```
java.sql.Statement stmt = conn.createStatement();
ResultSet r = stmt.executeQuery("SELECT x FROM Table2");
// Now retrieve the column 1 results in 4 K chunks:
byte buff = new byte[4096];
while (r.next()) {
       Java.io.InputStream fin = r.getAsciiStream(1);
       for (;;) {
              int size = fin.read(buff);
              if (size == 0) {
                     break;
              }
              // Send the newly filled buffer to some output stream:
              output.write(buff, 0, size);
       }
}
```

7.2.3 Optional or multiple ResultSets

Under some circumstances an application may not know whether a given statement will return a ResultSet until the statement has executed. In addition, some stored procedures may return several different ResultSets.

To accommodate these needs we provide a method java.sql.Statement.getMoreResults that can be used to obtain ResultSets from a statement.

We provide two distinct methods for executing a SQL statement, “execute” and “executeQuery” (see Section 13). “ExecuteQuery” should be used when it is known that the statement returns a single ResultSet. If a statement may return multiple ResultSets or if it is not known whether it will return a ResultSet, then we recommend using “execute” and then using “getMoreResults” to try to retrieve any ResultSets.

7.3 Passing statement parameters

7.3.1 Passing parameters IN

To allow you to pass parameters to a SQL statement, the java.sql.PreparedStatement class provides a series of setThing methods. These can be used before each statement execution to fill in parameter fields. Once a parameter value has been defined for a given statement, it can be used for multiple executions of that statement, until it is cleared by a call on PreparedStatement.clearParameters.

```
java.sql.PreparedStatement stmt = conn.prepareStatement(
                                  "UPDATE table3 SET m = ? WHERE x = ?");
// We pass two parameters. One varies each time around
// the for loop, the other remains constant.
stmt.setStringParameter(1, "Hi");
for (int i = 0; i < 10; i++) {
       stmt.setIntParameter(2, i);
       int rows = stmt.execute();
}
```

In an earlier version of the design we used method overloading so that rather than having methods with different names suach as setTinyInt, setBit, etc., all these methods were simply called setParameter, and were distinguished only by their different argument types. While this is a legal thing to do in Java, several reviewers commented that it was confusing and was likely to lead to error, particularly in cases where the mapping between SQL types and Java types is ambiguous. On reflection we agreed with them.

7.3.2 Receiving OUT parameters

If you are executing a stored procedure call that has OUT or INOUT parameters then you should use the CallableStatement class.

If you want to receive OUT parameters from the stored procedure then you must use the registerOutParameter method to register the SQL types of the OUT parameters before you execute the statement. Then after the statement has executed you can use one of the CallableStatement.get methods to retrieve the parameter value(s).

```
java.sql.CallableStatement stmt = conn.prepareCall(
                                  "{call getTestData(?, ?)}");
stmt.registerOutParameter(java.sql.Types.TINYINT);
stmt.registerOutParameter(java.sql.types.DECIMAL, 2);
for (int i = 0; i < 10; i++) {
       int rows = stmt.execute();
       byte x = stmt.getTinyIntParameter(1);
       Numeric n = stmt.getDecimal(2,2);
}
```

We dislike the need for a registerOutParameter method. During the development of JDBC we made a determined attempt to eradicate it and instead proposed that the drivers should use database metadata to determine the OUT parameter types. However reviewer input convinced us that for perfomance reasons it was more appropriate to require the use of registerOutParameter to specify OUT parameter types.

7.3.3 Sending very large parameters

JDBC itself defines no limits on the amount of data that may be sent with a setLongVarBinary or setLongVarChar call. However, when dealing with large blobs, it may be convenient for application programmers to pass in very large data in smaller chunks.

To accommodate this, we allow programmers to supply Java io streams as parameters. When the statement is executed the JDBC run-times will make repeated calls on these IO streams to read their contents and transmit these as the actual parameter data.

When setting a stream as an input parameter, the application programmer must specify the amount of data to be read from the stream and sent to the database.

We dislike requiring that the data transfer size be specified in advance, However this is necessary because some databases need to know the total transfer size in advance of any data being sent.

An example of using a stream to send the contents of a file as an IN parameter:

```
java.io.File file = new java.io.File("/tmp/foo");
int fileLength = file.length();
java.io.InputStream fin = new java.io.FileInputStream(fin);
java.sql.PreparedStatement stmt = conn.createPreparedStatement(
                  "UPDATE Table5 SET stuff = ? WHERE index = 4");
stmt.setBinaryStreamParameter(1, fin, fileLength);
// When the statement executes, the "fin" object will get called
// repeatedly to deliver up its data.
stmt.execute();
```

7.3.4 Receiving very large out parameters

We do not provide a mechanism for receiving very large values as OUT parameters.

Instead we recommend that programmers retrieve very large values through ResultSets.

If we were to provide a mechanism for very large OUT parameters, it would probably consist of allowing programmers to register java.io.OutputStreams into which the JDBC runtimes could send the OUT parameters data when the statement excutes. However this seems to be harder to explain than it is worth, given that there is a already a mechanism for handling large results as part of ResultSets.

8 Mapping SQL data types into Java

8.1 Constraints

We are required to provide reasonable Java mappings for the common SQL data types. We also need to make sure that we have enough type information so that we can correctly store and retrieve parameters and recover results from SQL statements

However, there is no particular reason that the Java data type needs to be exactly isomorphic to the SQL data type. For example, since Java has no fixed length arrays, we can represent both fixed length and variable length SQL arrays as variable length Java arrays. We also felt free to use Java Strings even though they don't precisely match any of the SQL CHAR types.

SQL type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	java.io.InputStream
NUMERIC	java.sql.Numeric
DECIMAL	java.sql.Numeric
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	float
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	java.sql.InputStream
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Table 1: Mapping from SQL types to Java types.
This table shows the standard Java types for mapping various common SQL types.

Table 1 shows the recommended Java mapping for various common SQL data types. The various mappings are described more fully in the following sections.

When retrieving result data JDBC will perform some coercions between SQL types. These coercions are shown in Table 2. Note that some coercions may result in loss of data or loss of precision.

	TINYINT	**SMALLINT**	**INTEGER**	**BIGINT**	**REAL**	**FLOAT**	**DOUBLE**	**DECIMAL**	**NUMERIC**	**BIT**	**CHAR**	**VARCHAR**	**LONGVARCHAR**	**BINARY**	**VARBINARY**	**LONGVARBINARY**	**DATE**	**TIME**	**TIMESTAMP**
getTinyInt	**X**	x	x	x	x	x	x	x	x	x	x	x	x						
getSmallInt	x	**X**	x	x	x	x	x	x	x	x	x	x	x						
getInteger	x	x	**X**	x	x	x	x	x	x	x	x	x	x						
getBigInt	x	x	x	**X**	x	x	x	x	x	x	x	x	x						
getFloat	x	x	x	x	**X**	x	x	x	x	x	x	x	x						
getReal	x	x	x	x	x	**X**	x	x	x	x	x	x	x						
getDouble	x	x	x	x	x	x	**X**	x	x	x	x	x	x						
getDecimal	x	x	x	x	x	x	x	**X**	x	x	x	x	x						
getNumeric	x	x	x	x	x	x	x	x	**X**	x	x	x	x						
getBit	x	x	x	x	x	x	x	x	x	**X**	x	x	x						
getChar	x	x	x	x	x	x	x	x	x	x	**X**	x	x	x	x	x	x	x	x
getVarChar	x	x	x	x	x	x	x	x	x	x	x	**X**	x	x	x	x	x	x	x
getLongVarChar	x	x	x	x	x	x	x	x	x	x	x	x	**X**	x	x	x	x	x	x
getBinary	x	x	x	x	x	x	x	x	x	x	x	x	x	**X**	x	x	x	x	x
getVarBinary	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**X**	x	x	x	x
getLongVarBinary	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**X**	x	x	x
getDate											x	x	x				**X**		x
getTime											x	x	x					**X**	x
getTimestamp											x	x	x				x		**X**
getAsciiStream											x	x	**X**	x	x	x			
getUnicodeStream											x	x	**X**	x	x	x			
getBinaryStream											x	x	x	x	x	**X**			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 2: Use of “get” methods to retrieve SQL data types.
An “x” means that the given “get” method can be used to retrieve the given SQL data type.
An “X” means that the given “get” method is recommended for retrieving the given SQL data type.

8.2 Dynamic data access

This chapter focuses on access to results or parameters whose types are known at compile time. However some applications, for example generic browsers or query tools, are not compiled with knowledge of the database schema they will access so JDBC also provides support for fully dynamically typed data access. See Section 14.2.

8.3 CHAR, VARCHAR, and LONGVARCHAR

There is no need for Java programmers to distinguish between the three different flavours of SQL strings CHAR, VARCHAR, and LONGVARCHAR. These can all be expressed identically in Java. It is possible to read and write the SQL correctly without needing to know the exact data type that was expected.

These types could be mapped to either String or char[]. After considerable discussion we decided to use String, as this seemed the more appropriate type for normal use. Note that the Java String class provides a method for converting a String to a char[] and a constructor for turning a char[] into a String.

The ResultSet.getChar, ResultSet.getVarChar, and ResultSet.getLongVarChar methods allocate and return a new String. This is suitable for retrieving normal data, but the LONGVARCHAR SQL type can be used to store multi-megabyte strings. We therefore needed to provide a way for Java programmers to retrieve a LONGVARCHAR value in chunks. We handle this by allowing programmers to retrieve a LONGVARCHAR as a Java input stream from which they can subsequently read data in whatever chunks they prefer. Java streams can be used for either Unicode or Ascii data, so the programmer may chose to use either getAsciiStream or getUnicodeStream.

8.4 DECIMAL and NUMERIC

The SQL DECIMAL and NUMERIC data types are used to express fixed point numbers where absolute precision is required. They are often used for currency values.

These two types can be expressed identically in Java.

We had some difficulty deciding on a suitable Java mapping. The most natural mapping would be to use a Java float or double. However because DECIMAL and NUMERIC are often used for currency values, it is considered unacceptable to use approximate floating point forms.

We allow access to DECIMAL and NUMERIC as simple Strings and arrays of chars.[1] Thus a Java programmers can use getString or getChars to receive a NUMERIC or DECIMAL result.

However this makes the common case of currency values rather awkward, as application writers now have to perform math on strings.

We therefore added a new type java.sql.Numeric that can be used to hold the SQL DECIMAL and NUMERIC types. This type is a subtype of the standard java.lang.Number type and provides math operations to allow Numeric types to be added, subtracted, multiplied, and divided with other Numeric types, with integer types, and with floating point types.

1. This is consistent with ODBC's treatment of these types.

8.5 BINARY, VARBINARY, and LONGVARBINARY

There is no need for Java programmers to distinguish between the three different flavours of SQL byte arrays BINARY, VARBINARY, and LONGVARBINARY. These can all be expressed identically as byte arrays in Java. (It is possible to read and write the SQL correctly without needing to know the exact BINARY data type that was expected. (**Confirm.**)

As with the LONGVARCHAR SQL type the LONGVARBINARY SQL type can sometimes be used to return multi-megabyte data values. We therefore allow a LONGVARBINARY value to be retrieved as a Java input stream , from which they can subsequently read data in whatever chunks they prefer.

8.6 BIT

The SQL BIT type can be mapped directly to the Java boolean type.

8.7 TINYINT, SMALLINT, INTEGER, and BIGINT

The SQL TINYINT, SMALLINT, INTEGER, and BIGINT types represent 8 bit, 16 bit, signed 32 bit, and 64 bit values. These therefore can be mapped to Java's byte, short, int, and long data types.

8.8 REAL, FLOAT, and DOUBLE

SQL defines three floating point data types, REAL, FLOAT, and DOUBLE.

We map REAL and FLOAT to Java float, and DOUBLE to Java double.

8.9 DATE, TIME, and TIMESTAMP

SQL defines three time related data types. DATE consists of day, month, and year. TIME consists of hours, minutes and seconds. TIMESTAMP combines DATE and TIME and also adds in a nanosecond field.

There is a standard Java class java.util.Date that provides data and time information. However this class doesn't perfectly match any of the three SQL types, as it includes both DATE and TIME information, but lacks the nanosecond granularity required for TIMESTAMP.

We therefore define three subclasses of java.util.date. These are:

- java.sql.Date for SQL DATE information
- java.sql.Time for SQL TIME information
- java.sql.Timestamp for SQL TIMESTAMP information

In the case of java.sql.Date the hour, minute, second, and milli-seconds fields of the java.util.Date base class are set to zero.

In the case of java.sql.time the year, month, and day fields of the java.util.Date base class are set to 1970, January, and 1 respectively. This is the "zero" date in the Java epoch.

The java.sql.timestamp class extends java.util.date by adding a nanosecond field.

9 Asynchrony, Threading, and Transactions

9.1 Asynchronous requests

Some database APIs, such as ODBC, provide mechanisms for allowing SQL statements to execute asynchronously. This allows an application to start-up a database operation in the background, and then handle other work (such as managing a user interface) while waiting for the operation to complete.

Since Java is a multi-threaded environment there seems no real need to provide support for asynchronous statement execution. The Java programmer can easily create a separate thread if they wish to execute statements asynchronously with respect to their main thread.

9.2 Multi-threading

We require that all operations on all the java.sql objects be multi-thread safe and can cope correctly with having several threads simultaneously calling the same object.

In practice we expect that most of the JDBC objects will only be accessed in a single threaded way. However some multi-thread support is necessary, and our attempts in previous drafts to specify some classes as MT safe and some as MT unsafe appeared to be adding more confusion than light.

Implementation note: Drivers can implement this by making all their methods "synchronized".

9.3 Transactions.

We provide an API at the connection level to allow programmers to begin a transaction (via the beginTransaction method) and then either commit or abort the transaction (via the commit or rollback methods). The transaction will cover all statement executions on that connection between the beginTransaction and the commit.

The exact semantics of transactions and their isolation levels depend on the underlying database. There are methods on java.sql.DatabaseMetaData to learn the current defaults, and on java.sql.Connection to move a newly opened connection to a different isolation level.

When a transaction is committed or aborted, all open statements on the connection will be closed.

ODBC leaves it up to the driver and database to decide whether or not to close statements after a commit or abort. We felt that we had to specify a clear default for such a key piece of behavior. However we also provide the Connection.disableAutoClose method to disable this default, and there are a set of DatabaseMetaData methods (supportsOpenStatementAcrossCommit, etc.) for determining what state the current database can preserve across a commit or rollback.

In the first revision of the interface we will provide no support for committing transactions across different connections.[1]

1. Again, this is consistent with ODBC.

10 Cursors

JDBC provides simple cursor support. An application can use ResultSet.getCursorName() to obtain a cursor associated with the current ResultSet. It can then use this cursor name in positioned update or positioned delete statements.

(Note that not all databases support positioned update and delete. The DatabaseMetaData.supportsPositionedDelete and supportsPositionedUpdate methods can be used to discover whether a particular connection supports these operations.)

The cursor will remain valid until the ResultSet or its parent Statement is closed.

Currently we do not propose to provide support for either scrollable cursors or ODBC style bookmarks as part of JDBC.

11 Escape for Extended SQL

Certain SQL features, specifically stored procedures, scalar functions, dates, times, and outer joins, are widely supported by DBMSs, but DBMSs use different syntax for the same functionality. JDBC supports the same DBMS-independent "escape" syntax as ODBC for these features. A driver can then map this syntax into DBMS-specific syntax, allowing portability of application programs that require these features. The DBMS-independent syntax is based on an escape clause demarcated by curly braces and a keyword:

```
{keyword ... parameters ...}
```

11.1 Stored Procedures

The syntax for invoking a stored procedure in JDBC is:

```
{call procedure_name[argument1, argument2, ...]}
```

or, where a procedure returns a result parameter:

```
{?= call procedure_name[argument1, argument2, ...]}
```

Input arguments may be either literals or parameters.

11.2 Time and Date Syntax

DBMSs differ in the syntax they use for dates, times, and timestamps. JDBC supports ISO standard format for the syntax of these literals, using an escape clause that the driver must translate to the DBMS representation.

For example, a date is specified in a JDBC SQL statement with the syntax

```
{d 'yyyy-mm-dd'}
```

where yyyy-mm-dd provides the year, month, and date, e.g. 1996-02-28. The driver will replace this escape clause with the equivalent DBMS-specific representation, e.g. 'Feb 28, 1996' for Oracle.

There are analogous escape clauses for TIME and TIMESTAMP:

```
{t 'hh:mm:ss'}

{ts 'yyyy-mm-dd hh:mm:ss.f...'}
```

11.3 Scalar Functions

JDBC supports numeric, string, time, date, system, and conversion functions on scalar values. These functions are indicated by the keyword "fn" followed by the name of the desired function and its arguments. For example, two strings can be concatenated using the concat function

```
{fn concat("Hot", "Java")}
```

The name of the current user can be obtained through the syntax

```
{fn user()}
```

See the X/Open CLI or ODBC specifications for specifications of the semantics of the scalar functions. The functions supported are listed here for reference. Some drivers may not support all of these functions; to find out which functions are supported, you can call the metadata interfaces described in Section 15: getNumericFunctions() returns a comma separated list of the names of the numeric functions supported, getStringFunctions() does the same for the string functions, and so on.

The numeric functions are ABS(number), ACOS(float), ASIN(float), ATAN(float), ATAN2(float1, float2), CEILING(number), COS(float), COT(float), DEGREES(number), EXP(float), FLOOR(number), LOG(float), LOG10(float), MOD(integer1, integer2), PI(), POWER(number, power), RADIANS(number), RAND(integer), ROUND(number, places), SIGN(number), SIN(float), SQRT(float), TAN(float), and TRUNCATE(number, places).

The string functions are ASCII(string), CHAR(code), CONCAT(string1, string2), DIFFERENCE(string1, string2), INSERT(string1, start, length, string2), LCASE(string), LEFT(string, count), LENGTH(string), LOCATE(string1, string2, start), LTRIM(string), REPEAT(string, count), REPLACE(string1, string2, string3), RIGHT(string, count), RTRIM(string), SOUNDEX(string), SPACE(count), SUBSTRING(string, start, length), and UCASE(string).

The time and date functions are CURDATE(), CURTIME(), DAYNAME(date), DAYOFMONTH(date), DAYHOFWEEK(date), DAYOFYEAR(date), HOUR(time), MINUTE(time), MONTH(time), MONTHNAME(date), NOW(), QUARTER(date), SECOND(time), TIMESTAMPADD(interval, count, timestamp), TIMESTAMPDIFF(interval, timestamp1, timpestamp2), WEEK(date), and YEAR(date).

The system functions are DATABASE(), IFNULL(expression, value), and USER().

There is also a CONVERT(value, SQLtype) expression, where type may be BIGINT, BINARY, BIT, CHAR, DATE, DECIMAL, DOUBLE, FLOAT, INTEGER, LONGVARBINARY, LONGVARCHAR, REAL, SAMALLINT, TIME, TIMESTAMP, TINYINT, VARBINARY, and VARCHAR.

Again, these functions are supported by DBMSs with slightly different syntax, and the driver's job is to either map these into the appropriate syntax or to implement the function directly in the driver.

11.4 Outer Joins

The syntax for an outer join is

```
{oj outer-join}
```

where outer-join is of the form

```
table LEFT OUTER JOIN {table | outer-join} ON search-condition
```

See the SQL grammar for an explanation of outer joins.

12 Variants and Extensions

As far as possible we would like a standard JDBC API that works in a uniform way with different databases. However it is unavoidable that different databases support different SQL features, and provide different semantics for some operations.

12.1 Permitted variants

The methods in java.sql.Connection, java.sql.Statement, java.sql.PreparedStatement, and java.sql.Resultset should all be supported for all JDBC drivers.

However, the actual SQL that may be used varies somewhat between databases. For example, different databases provide different support for outer joins. The java.sql.DatabaseMetaData interface provides a number of methods that can be used to determine exactly which SQL features are supported by a particular database. Similarly, the syntax for a number of SQL features may vary between databases and can also be discovered from java.sql.DatabaseMetaData. However, we require at least ANSI SQL-2 Entry Level syntax and semantics in order to pass JDBC conformance tests.

Finally, some fundamental properties such as transaction isolation vary between databases. The default properties of the current database, and the range of properties it supports, can also be obtained from java.sql.DatabaseMetaData.

12.2 Vendor specific extensions

JDBC provides a uniform API that is intended to work across all databases. However, database vendors may wish to expose additional functionality that is supported by their databases.

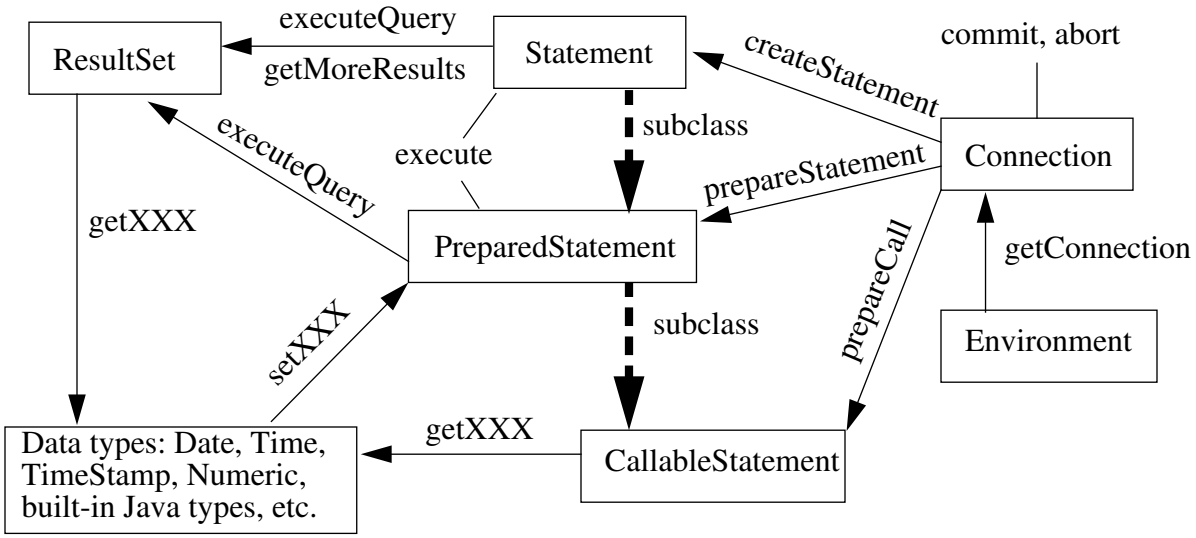
Database vendors may add additional functionality by adding new subtypes of existing JDBC types that provide additional methods. Thus the Foobah corporation might define a new Java type foobah.sql.FooBahStatement that inherits from the standard java.sql.Statement type but adds some new functionality.

13 JDBC Interface Definitions

The following pages contain the Java definitions of the core java.sql interfaces and classes:

java.sql.CallableStatement
java.sql.Connection
java.sql.Date
java.sql.Driver
java.sql.Environment
java.sql.Numeric
java.sql.PreparedStatement
java.sql.ResultSet
java.sql.Statement
java.sql.SQLException
java.sql.Time
java.sql.Timestamp
java.sql.Types

A later chapter, Chapter 15, provides definitions of the JDBC metadata interfaces; see also the short example programs in Appendix B. The more important relationships between the interfaces are as follows (with arrows showing functions and lines showing other methods):



```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// A CallableStatement object is used to execute a stored procedure call.
//
// See also the PreparedStatement baseclass, which privides methods for
// setting IN parameters, and the findamental Statement baseclass.

package java.sql;

public interface CallableStatement extends PreparedStatement {

    // Before executing a stored procedure call you must explicitly
    // call registerOutParamter to register the java.sql.Type of each
    // out parameter.
    void registerOutParameter(int parameterIndex, int sqlType)
                            throws SQLException;
    // You must also specify the scale for numeric/decimal types:
    void registerOutParameter(int parameterIndex, int sqlType, int scale)
                            throws SQLException;

    boolean isNull(int parameterIndex) throws SQLException;

    // Methods for retrieving OUT parameters from this statement.
    String getChar(int parameterIndex) throws SQLException;
    String getVarChar(int parameterIndex) throws SQLException;
    String getLongVarChar(int parameterIndex) throws SQLException;
    boolean getBit(int parameterIndex) throws SQLException;
    byte getTinyInt(int parameterIndex) throws SQLException;
    short getSmallInt(int parameterIndex) throws SQLException;
    int getInteger(int parameterIndex) throws SQLException;
    long getBigInt(int parameterIndex) throws SQLException;
    float getReal(int parameterIndex) throws SQLException;
    float getFloat(int parameterIndex) throws SQLException;
    double getDouble(int parameterIndex) throws SQLException;
    Numeric getNumeric(int parameterIndex, int scale)
                            throws SQLException;
    Numeric getDecimal(int parameterIndex, int scale)
                            throws SQLException;
    byte[] getBinary(int parameterIndex) throws SQLException;
    byte[] getVarBinary(int parameterIndex) throws SQLException;
    byte[] getLongVarBinary(int parameterIndex) throws SQLException;

    java.sql.Date getDate(int parameterIndex) throws SQLException;
    java.sql.Time getTime(int parameterIndex) throws SQLException;
    java.sql.Timestamp getTimestamp(int parameterIndex)
                            throws SQLException;

    // Advanced features:

    // You can obtain a ParamterMetaData object to get information
    // about the parameters to this CallableStatement.
    ParameterMetaData getMetaData();

    // getObject returns a Java object for the parameter.
```

```
        // See the JDBC spec's "Dynamic Programming" chapter for details.
        Object getObject(int parameterIndex, int sqlType)
                                throws SQLException;

    }
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

public interface Connection {

    // createStatement returns an SQL Statement object
    Statement createStatement() throws SQLException;

    // prepareStatement creates a pre-compiled SQL PreparedStatement object.
    PreparedStatement prepareStatement(String sql)
                        throws SQLException;

    // prepareCall create a pre-compiled SQL statement that is
    // a call on a stored procedure.
    CallableStatement prepareCall(String sql) throws SQLException;

    // Convert the given generic SQL statement to the drivers native SQL.
    String nativeSQL(String query) throws SQLException;

    // If "autoCommit" is true, then all subsequent SQL statements will
    // be executed and committed as individual transactions.  Otherwise
    // (if "autoCommit" is false) then subsequent SQL statements will
    // all be part of the same transaction , which must be expicitly
    // committed with either a "commit" or "rollback" call.
    // By default new connections are initialized with autoCommit "true".
    void setAutoCommit(boolean enableAutoCommit) throws SQLException;

    // You can use commit or rollback to commit or abort a transaction.
    // Note that by default all active Statement or ResultSet objects
    // associated with a Connection will be closed when it is committed
    // or aborted.

    void commit() throws SQLException;
    void rollback() throws SQLException;

    // close frees up various state associated with the connection.
    void close() throws SQLException;;

    // isClosed returns true if the connection is closed, which can
    // occur either due to an explicit call on "close" or due to
    // some fatal error on the connection.
    boolean isClosed() throws SQLException;;

    //======================================================================
    // Advanced features:

    // You can obtain a DatabaseMetaData object to get information
    // about the target database.
    DatabaseMetaData getMetaData() throws SQLException;;

    // You can put a connection in readonly mode as a hint to enable
    // database optimizations.  Note that setReadOnly cannot be called
    // while in the middle of a transaction.
```

```
    void setReadOnly(boolean readOnly) throws SQLException;
    boolean isReadOnly() throws SQLException;

    // The "catalog" selects a sub-space of the target database.
    void setCatalog(String catalog) throws SQLException;
    String getCatalog() throws SQLException;

    // You can call the following method to try to change the transaction
    // isolation level on a newly opened connection, using one of the
        // TRANSACTION_* values.  Use the DatabaseMetaData class to find what
    // isolation levels are supported by the current database.
    // Note that setTransactionIsolation cannot be called while in the
    // middle of a transaction.
    int TRANSACTION_NONE = 0;
    int TRANSACTION_READ_UNCOMMITTED = 1;
    int TRANSACTION_READ_COMMITTED   = 2;
    int TRANSACTION_REPEATABLE_READ  = 3;
    int TRANSACTION_SERIALIZABLE     = 4;
    int TRANSACTION_VERSIONING       = 5;
    void setTransactionIsolation(int level) throws SQLException;
    int getTransactionIsolation() throws SQLException;

    // disableAutoClose can be used on a newly opened connection to to
    // disable the automatic closing of Statements and ResultSets during
    // calls on Commit and Rollback.  This functionlity is only available
    // in some databases, and the exact semantics of the objects state
    // vary between databases.  See the DatabaseMetadatData class for
    // methods to learn about the consequences of doing disableAutoClose.
    void disableAutoClose() throws SQLException;

    // getWarnings will return any warning information related to
    // the current connection.  Note that SQLWarning may be a chain.
    SQLWarning getWarnings() throws SQLException;
    void clearWarnings() throws SQLException;
}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// The DataTrunction class is a SQLWarning that is added to a Statement
// when data truncation occurs during a database access.

package java.sql;

public class DataTruncation extends SQLWarning {

    public DataTruncation();

    // Index of column or parameter that was truncated:
    public int index;

    // Was it a parameter?
    public boolean parameter = false;

    // Was it truncated on a read from the database?
    public boolean read = true;

    // Amount of data that should have been transferred.  This
    // number may be approximate if data conversions were being
    // performed.
    public int dataSize;
    // Number of bytes of data actually transferred.
    public int transferSize;

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// This class is used to represent a subset of the standard java.util.date
// information.  We only deal with daya and ignore hours, minutes, and seconds.
// This lets us represent SQL DATE information.

package java.sql;

public class Date extends java.util.Date {

    public Date(int year, int month, int day);

    public String toString ();

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// The Java SQL framework allows for multiple database drivers.
//
// Each driver should supply a Driver class that implements the java.sql.Driver
// public interface.
//
// As part of the SQL framework, the framework will try to load as many drivers
// as it can find and then for any given connection request it will ask each driver
// in turn to try to connect to the target URL.
//
// It is strongly recommended that each Driver class should be small and standalone
// so that the Driver class can be loaded and queried without bringing in vast
// quantities of supporting code.
//
// When a Driver object is instantaiated it should register itself with the SQL
// framework by caling java.sql.Environment.registerDriver


package java.sql;

public interface Driver {

    // Each Driver should also support a null constructor so it can be
    // instantiated by doing
    //      java.sql.Driver d = Class.forName("foo.bah.Driver").newInstance();


    // Try to make a database connection to the given URL.
    // The driver should return "null" it it realizes it is the wrong kind
    // if driver to connect to the given URL.  This will be common, as when
    // the JDBC driver manager is asked to connect to a given URL it passes
    // the URL to each loaded driver in turn.
    //
    // The driver should raise a SQLException if it is the right
    // driver to connect to the given URL, but has trouble connecting to
    // the database.
    //
    // The java.util.Properties argument can be used to passed arbitrary
    // string tag/value pairs as connection arguments.
    // Normally at least a "user" and  "password" properties should be
    // included in the Properties.

    Connection connect(String url, java.util.Properties info)
                        throws SQLException;

    // majorVersion and minorVersion return the major and minor
    // versions of the driver.  Initially these should be 1 and 0.
    int majorVersion();
    int minorVersion();
}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// The java.sql.Environment class provides access to global SQL state.
//
// As part of its initialization the Envioronment class will use the system
// "jdbc.drivers" property.  This should contain a list of classnames for
// driver classes.  The Environment class wil attempt to load each of these
// driver classes.  For example in you ~/.hotjava/properties file you might
// specify:
//  jdbc.drivers=foo.bah.Driver wombat.sql.Driver bad.taste.ourDriver
//
// Sunsequently when you attempt to open a database connection the Environment
// class will attempt to locate a suitable driver class from amongst these
// driver classes and from any other driver classes which have been loaded
// from the same classloader as the current applet.

package java.sql;

public class Environment {

    // The getConnection function attempts to establish a connection to the
    // given database URL.  It will attempt to select an approrpiate driver
    // from the set of registered SQL drivers.
    // The user and password fields are passed to the driver which may
    // use them to connect to the server.  The driver may also use other
    // authentication information from the current applet environment.

    public static synchronized Connection getConnection(String url,
            java.util.Properties info) throws SQLException;

    public static synchronized Connection getConnection(String url,
            String user, String password) throws SQLException;

    // A newly loaded driver class should call registerDriver to make itself
    // known to the SQL framework.
    public static synchronized void registerDriver(java.sql.Driver driver)
                        throws SQLException;

    // An applet can call "loadDriver" to attempt to load an additional named
    // driver class.  The result is true if the load succeeded.
    public static boolean loadDriver(String classname);

    // The getDrivers method returns an Enumeration of all the JDBC drivers
    // currently loaded.  (Note however that it may still be possible to load
    // other drivers using Environment.loadDriver.)
    // The classname of a driver can be found using d.getClass().getName();

    public static java.util.Enumeration getDrivers();


    // This defines a maximum timeout in seconds that drievrs should use
    // when attempting to login to remote databases.
    void setLoginTimeout(int seconds);
    int getLoginTimeout();
```

```
        // This defines a logging/trace Stream that should be used by the JDBC
        // manager and by the drivers.  You can use "null" to disable tracing.
        void setLogStream(java.io.PrintStream out);
        java.io.PrintStream getLogStream();


    }
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// The NullData class is a SQLWarning that is added to a Statement
// when a SQL NULL valeu is read during dtabase access.

package java.sql;

public class NullData extends SQLWarning {

    public NullData();

    // Index of column or parameter that was read as NULL:
    public int index;

    // Was it a parameter?
    public boolean parameter = false;

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// The Numeric class can be used to represent SQL fixed point
// NUMERIC and DECIMAL values.  The "scale" of a Numeric object is the
// number of digits of precision it supports to the right of the decimal.
//
// Numeric objects are immutable.  The math operations return new objects.
//
// A Numeric value consists of up to 18 digits.  The scale (meaning the number
// of digits to the right of the decimal) may be from 0 to 9.
//
//                              KGH Dec 95

package java.sql;

public final class Numeric extends java.lang.Number {

    // scale specifies how many decimal digits after the floating point
    // we should maintain.

    public Numeric(String s, int scale);

    public Numeric(int x, int scale);

    public Numeric(double x, int scale);

    public Numeric(Numeric x);

    public Numeric(Numeric x, int scale);

    // The following methods convert the Numeric value to an appropiate Java
    // built-in type.  Note that this may involve loss of precision.
    public int intValue();

    public long longValue();

    public float floatValue();

    public double doubleValue();

    public String toString();

    public int getScale();

    // getScaled returns the value multipied by 10**scale.
    // Thus if a currency value was being kept with scale "2" as "5.04",
    // the getScaled function would return the long integer "504".
    public long getScaled();

    // create from scaled takes a scaled value and turns it into a
    // numeric value by dividing the scaled value by 10**scale.
    // Thus createFromScaled(504,2) would create the value "5.04"
    public static Numeric createFromScaled(long scaled, int s);
```

```
    // The following math methods can be used to create new Numeric objects.
    public Numeric add(Numeric x);

    public Numeric subtract(Numeric x);

    public Numeric multiply(Numeric x);

    public Numeric divide(Numeric x);

    public boolean equals(Object obj);

    public boolean lessThan(Numeric x);

    public boolean lessThanOrEquals(Numeric x);

    public boolean greaterThan(Numeric x);

    public boolean greaterThanOrEquals(Numeric x);

    public int hashCode();



}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// A PreparedStatement object is used to execute a pre-compiled SQL statement.
//
// See also the Statement base-type, which can execute ad-hoc SQL statement,
// and the CallStatement sub-type which can be used for calls on stored
// procedures.

package java.sql;

public interface PreparedStatement extends Statement {

    // This method excutes a "prepared" query statement.
    ResultSet executeQuery() throws SQLException;

    // This method executed a "prepared" statement that modifies the database.
    // The result is the number of rows updated.
    int execute() throws SQLException;

    // Methods for setting IN parameters into this statement.
    // Parameters are numbered starting at 1.

    // You always need to specify a SQL type when sending a NULL.
    void setNull(int parameterIndex, int sqlType) throws SQLException;

    // The following methods allow you to set various SQLtypes as parameters.
    // Note that the method include the SQL type in their name.
    void setBit(int parameterIndex, boolean x) throws SQLException;
    void setTinyInt(int parameterIndex, byte x) throws SQLException;
    void setSmallInt(int parameterIndex, short x) throws SQLException;
    void setInteger(int parameterIndex, int x) throws SQLException;
    void setBigInt(int parameterIndex, long x) throws SQLException;
    void setReal(int parameterIndex, float x) throws SQLException;
    void setFloat(int parameterIndex, float x) throws SQLException;
    void setDouble(int parameterIndex, double x) throws SQLException;

    void setNumeric(int parameterIndex, Numeric x)
                            throws SQLException;
    void setDecimal(int parameterIndex, Numeric x)
                            throws SQLException;

    void setChar(int parameterIndex, String x) throws SQLException;
    void setVarChar(int parameterIndex, String x) throws SQLException;
    void setLongVarChar(int parameterIndex, String x)
                            throws SQLException;

    void setBinary(int parameterIndex, byte x[]) throws SQLException;
    void setVarBinary(int parameterIndex, byte x[]) throws SQLException;
    void setLongVarBinary(int parameterIndex, byte x[])
                            throws SQLException;

    void setDate(int parameterIndex, java.sql.Date x)
                            throws SQLException;
    void setTime(int ParameterIndex, java.sql.Time x)
```

```
                            throws SQLException;
    void setTimestamp(int ParameterIndex, java.sql.Timestamp x)
                            throws SQLException;

    // The normal setChars and setBinary methods are suitable for passing
    // normal sized data.  However occasionally it may be necessary to send
    // extremely large values as LONGVARCHAR or LONGVARBINARY parameters.
    // In this case you can pass in a java.io.InputStream object, and the
    // JDBC runtimes will read data from that stream as needed, until they reach
    // end-of-file.  Note that these stream objects can either be standard Java
    // stream objects, or your own subclass that implements the standard interface.
    // setAsciiStreamParameter and setUnicodeStreamParameter imply the use of
    // the SQL LONGVARCHAR type, and setBinaryStreamParameter implies the
    // SQL LONGVARBINARY type.
    // For each stream type you must specify the number of bytes to be read
    // from the stream and sent to the database.

    void setAsciiStream(int ParameterIndex, java.io.InputStream x, int length)
                            throws SQLException;
    void setUnicodeStream(int ParameterIndex, java.io.InputStream x, int length)
                            throws SQLException;
    void setBinaryStream(int ParameterIndex, java.io.InputStream x, int length)
                            throws SQLException;

    // Finally, if you need a more dynamic interface, or if you need
    // the driver to perform a supported coercion, then you can use
    // setObjectParameter.
    // You must explicitly specify the SQL type that you want.
    // The supported sub-types of Object are java.lang.Boolean,
    // java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long,
    // java.lang.Float, java.lang.Double, java.lang.String,
    // Numeric, java.io.inputStream, byte[],
    // java.util.Date, and their subtypes.  You can also use a Java null.

    // Parameters remain in force for repeated use of the same statement.
    // You can use clearParameters to remove any parameters associated with
    // a statement.
    void clearParameters() throws SQLException;

    // Advanced features:

    // You can set a parameter as a Java object.  See the JDBC spec's
    // "Dynamic Programming" chapter for information on valid Java types.
    void setObject(int ParameterIndex, Object x, int sqlType)
                            throws SQLException;

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// The ResultSet object maintains a cursor pointing at a row and provides a
// set of get* methods for accessing columns within the current row.  Initially
// the cursor is positioned before the first row.  The "next" method moves
// the cursor to the next row.
//
// You can use the ResultSet.get methods to retrieve column values for
// the current row of a ResultSet.  The columns are numbered starting at 1.
//
// Each ResultSet is associated with a Statement object.  The ResultSet
// will be automatically closed when the associated statement is either
// closed or re-executed.


package java.sql;

public interface ResultSet {
    // The isNull() method returns true if the given column in the
    // current row holds the SQL "null".
    boolean isNull(int column) throws SQLException;

    // Methods for accessing columns in the current row.
    String getChar(int column) throws SQLException;
    String getVarChar(int column) throws SQLException;
    String getLongVarChar(int column) throws SQLException;
    boolean getBit(int column) throws SQLException;
    byte getTinyInt(int column) throws SQLException;
    short getSmallInt(int column) throws SQLException;
    int getInteger(int column) throws SQLException;
    long getBigInt(int column) throws SQLException;
    float getFloat(int column) throws SQLException;
    float getReal(int column) throws SQLException;
    double getDouble(int column) throws SQLException;
    Numeric getNumeric(int column, int scale)
                    throws SQLException;
    Numeric getDecimal(int column, int scale)
                    throws SQLException;
    byte[] getBinary(int column) throws SQLException;
    byte[] getVarBinary(int column) throws SQLException;
    byte[] getLongVarBinary(int column) throws SQLException;
    java.sql.Date getDate(int column) throws SQLException;
    java.sql.Time getTime(int column) throws SQLException;
    java.sql.Timestamp getTimestamp(int column) throws SQLException;

    // The normal getChars and getBinary methods are suitable for
    // reading normal sized data.  However occasionally it may be
    // necessary to access LONGVARCHAR or LONGVARBINARY fields that
    // are multiple Megabytes in size.  To support this case we provide
    // getCharStream and getBinaryStream methods that return Java
    // IO streams from which data can be read in chunks.

    java.io.InputStream getAsciiStream(int column)
                            throws SQLException;
```

```
    java.io.InputStream getUnicodeStream(int column)
                            throws SQLException;
    java.io.InputStream getBinaryStream(int column)
                            throws SQLException;

    // next moves us to the next row of the results.
    // It returns true if if has moved to a valid row, false if
    // all the rows have been processed.
    boolean next() throws SQLException;

    // Return the number of the current row.  The first row containing
    // actual data is row 1.
    int getRowNumber() throws SQLException;

    // Return the number of columns in the ResultSet
    int getColumnCount() throws SQLException;

    // The total number of rows returned by the query.  Note that on some
    // databases this method may be very expensive.
    int getRowCount() throws SQLException;

    // close frees up internal state associated with the ResultSet
    // You should normally call close when you are done with a ResultSet
    // ResultSets are also closed automatically when their Statement is closed.
    void close() throws SQLException;

    // Advanced features:

    // getCursorname returns a cursor name that can be used to identify the
    // currnet position in the Resultset to a separate statement that is
    // executing a positioned update or positioned delete.  If the database
    // doesn't support positioned delete or update, it may return "" here.
    String getCursorName() throws SQLException;

    // You can obtain a ResultMetaData object to obtain information
    // about the number, types and proerties of the result columns.
    ResultSetMetaData getMetaData() throws SQLException;

    // You can obtain a column result as a Java Object.
    // See the JDBC spec's "Dynamic Programming" chapter for details.
    Object getObject(int column, int sqlType) throws SQLException;
}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.


// The SQLException class privides information on a database access
// error.
// Each SQLException provides several kinds of information:
//   (1) a string describing the error.  This is used as the Java Exception
//       message, and is available via the getMesage() method
//   (2) A "SQLstate" string which follows the XOPEN SQLstate  conventions.
//       The values of the SQLState string adesfribed in the XOPEN SQL spec.
//   (3) An integer error code that is vendor specific.  Normally this will
//   be the actual error code returned by the underlying database.
//   (4) A chain to a next Exception.  This can be used to provided additional
//   error information.


package java.sql;

public class SQLException extends java.lang.Exception {

    // This constructor takes a SQL error state, a descriptive String, and
    // a vendor specific error code.
    public SQLException(String reason, String SQLstate, int vendorCode);


    // This constructor takes an SQL error string and a descriptive String.
    public SQLException(String reason, String SQLstate);

    public SQLException(String reason);

    public SQLException();

    public String getSQLState();

    public int getErrorCode();

    // The getNextException and setNextExcpetion methods allow programmers
    // to chain lower-level additional exception information of an exception.

    public SQLException getNextException();

    public void setNextException(SQLException ex);

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

public class SQLWarning extends SQLException {

    public SQLWarning(String reason, String SQLstate, int vendorCode);


    public SQLWarning(String reason, String SQLstate);

    public SQLWarning(String reason);

    public SQLWarning();

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// A statement object is used to execute an SQL statement.
//
// See also the PreparedStatement subtype of Statement.


package java.sql;

public interface Statement {
    // This method excutes the given SQL query statement.
    ResultSet executeQuery(String query) throws SQLException;

    // This method executes the given SQL statement that modifies the
    // database.  This includes things like UPDATE, INSERT, DELETE, etc.
    // The result is the number of rows updated.
    // Note that if a stored procedure is being called then you can use
    // "execute" to get a rowCount and then also do getMoreResults
    // to retrieve one or more ResultSets.
    int execute(String sql) throws SQLException;

    // close frees up internal state associated with the statement.
    // You should normally call close when you are done with a statement.
    // N.B. any ResultSets associated with the Statement will become
    // unusable when the Statement is closed.
    void close();

    // Under some (uncommon) situations a single SQL statement may return
    // multiple result sets and/or update counts.  Normally you can ignore
    // this, unless you're executing a stored procedure that you know may
    // return multiple results.
    //
    // If a query returned multiple ResultSets then you can use getMoreResults()
    // to get subsequent ResultSets.  The result is null if there are no more
    // ResultSets or if the next result is an update count.
    //
    // If an update returned multiple count values, then you can use
    // getMoreCounts() to get subsequent counts.  The result is -1 if
    // there are no more counts or if the next result is a ResultSet.
    //
    // So to be sure you have processed all the results, you must wait until
    // getMoreResults returns null && getMoreCounts returns -1.

    ResultSet getMoreResults() throws SQLException;
    int getMoreCounts() throws SQLException;


    // The maxFieldSize is a limit (in bytes) on how much data can be
    // returned as part of a field.  If the limit is exceeded, the data
    // is truncated and a warning is added (see Statement.getWarnings).
    // You should normally not need to change the default limit.
    // Zero means no limit.
    int getMaxFieldSize() throws SQLException;
    void setMaxFieldSize(int max) throws SQLException;
```

```
    // The maxRows value is a limit on how many rows can be returned as
    // part of a ResutlSet.  You should normally not need to change
    // the default limit.  Zero means no limit.
    int getMaxRows() throws SQLException;
    void setMaxRows(int max) throws SQLException;

    // if escape scanning is on (the default) the driver will do escape
    // substitution before sending the SQL to the database.
    void setEscapeProcessing(boolean enable) throws SQLException;

    // The queryTimeout is how many seconds the driver will allow for an
    // SQL statement to execute before giving up.  Zero means no limit.
    int getQueryTimeout() throws SQLException;
    void setQueryTimeout(int seconds) throws SQLException;

    // Cancel can be used by one thread to cancel a statement that
    // is being executed by another thread.
    void cancel() throws SQLException;

    // getWarnings will return any warning information related to
    // the current statement execution.  Note that SQLWarning may be
    // a chain.  If a statement is re-executed then warnings associated
    // with its previous execution will be automatically discarded.

    SQLWarning getWarnings() throws SQLException;
    void clearWarnings() throws SQLException;

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// This class is used to represent a subset of the standard java.util.date
// information.  We only deal with hours, minutes, and seconds.
// This lets us represent SQL TIME information.

package java.sql;

public class Time extends java.util.Date {

    public Time(int hour, int minute, int second);

    public String toString ();

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// This class extends the stanadard sun.util.date class with nanos.
// This lets it represent SQL timestamps.

package java.sql;

public class Timestamp extends java.util.Date {

    public Timestamp(int year, int month, int date,
                int hour, int minute, int second, int nano);

    public int nanos;

    public String toString ();

}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// This class defines constants that are used to identify SQL types.
//
// The actual type constant values are equivalent to those in XOPEN.


package java.sql;

public class Types {

    public final static int BIT =  -7;
    public final static int TINYINT =  -6;
    public final static int SMALLINT=   5;
    public final static int INTEGER =   4;
    public final static int BIGINT =  -5;

    public final static int FLOAT =   6;
    public final static int REAL =   7;
    public final static int DOUBLE =   8;

    public final static int NUMERIC =   2;
    public final static int DECIMAL=   3;

    public final static int CHAR=   1;
    public final static int VARCHAR =  12;
    public final static int LONGVARCHAR =  -1;

    public final static int DATE =  91;
    public final static int TIME =  92;
    public final static int TIMESTAMP =  93;

    public final static int BINARY=  -2;
    public final static int VARBINARY =  -3;
    public final static int LONGVARBINARY =  -4;


    public final static int NULL =   0;
}
```

14 Dynamic database access

We expect most JDBC programmers will be programming with knowledge of their target database's schema. They can therefore use the strongly typed JDBC interfaces described in Section 7 for data access. However there is also another extremely important class of database access where an application (or an application builder) dynamically discovers the database schema information and uses that information to perform appropriate dynamic data access. This section and the Java interface definitions in Section 15 describe the JDBC support for dynamic access.

14.1 Metadata information

JDBC provides access to a number of different kinds of metadata, describing row results, statement parameters, database properties, etc., etc. We originally attempted to provide this information via extra methods on the core JDBC classes such as java.sql.connection and java.sql.ResultSet. However, because of the complexity of the metadata methods and because they are only likely to be used by a small subset of JDBC programmers, we decided to split the metadata methods off into these three separate Java interfaces.

In general, for each piece of metadata information we have attempted to provide a separate JDBC method that takes appropriate arguments and provides an appropriate Java result type. However, when a method such as Connection.getProcedures() returns a collection of values, we have chosen to use a java.sql.ResultSet to contain the results. The application programmer can then use normal ResultSet methods to iterate over the results.

We considered defining a set of enumeration types for retrieving collections of metadata results, but this seemed to add additional weight to the interface with little real value. JDBC programmers will already be familiar with using ResultSets, so using them for metadata results should not be too onerous.

A number of metadata methods take String search patterns as arguments. These search patterns are the same as for ODBC, where a "_" implies a match of any single character and a "%" implies a match of zero or more characters. A Java null String implies "match anything".

The java.sql.ResultSetMetaData type provides a number of methods for discovering the types and properties of the columns of a particular java.sql.ResultSet object.

The java.sql.ParameterMetaData type proves methods for discovering the types and properties of parameters associated with a particular java.sql.PreparedStatement

The java.sql.DatabaseMetadata interface provides methods for retrieving various metadata associated with a database. This includes enumerating the stored procedures in the database, the tables in the database, the schemas in the database, the valid table types, the valid catalogs, finding information on the columns in tables, access rights on columns, access rights on tables, minimal row identification, and so on.

14.2 Dynamically typed data access

In Section 8 we described the normal mapping between SQL types and Java types. For example, a SQL INTEGER is normally mapped to a Java int. This supports a simple interface for reading and writing SQL values as simple Java types.

However, in order to support generic data access, we also provide methods that allow data to be retrieved as generic Java objects. Thus there is a ResultSet.getObject method, a PreparedStatement.setObject method, and a CallableStatement.getObject method. For each of the two getObject methods you will need to narrow the resulting java.lang.Object object to a specific data type before you can retrieve a value.

Since the Java built-in types such as boolean and int are not subtypes of Object, we need to use a slightly different mapping from SQL types to Java object types for the getObject/setObject methods. This mapping is shown in Table 3.

SQL type	Java Object Type
CHAR	String
VARCHAR	String
LONGVARCHAR	java.io.InputStream
NUMERIC	java.sql.Numeric
DECIMAL	java.sql.Numeric
BIT	Boolean
TINYINT	Integer
SMALLINT	Integer
INTEGER	Integer
BIGINT	Long
REAL	Float
FLOAT	Float
DOUBLE	Double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	java.sql.InputStream
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Table 3: Mapping from SQL types to Java Object types.

Because of the overlap between the various get/set methods and the generic getObject/setObject methods we looked at discarding our get/set methods and simply using getObject/setObject. However for the simple common cases where a programmer know the SQL types, the resulting casts and extracts are extremely tedious:

```
int i = ((Integer)r.getObject(1)).intValue()
```

We therefore decided to bend our minimalist principles a little in this case and retain the various get/set methods as the preferred interface for the majority of applications programmers, while also adding the getObject/setObject methods for tool builders and sophisticated applications.

15 JDBC Metadata Interfaces

The following pages contain the Java definitions of the JDBC metadata interfaces:

java.sql.DatabaseMetaData
java.sql.ParameterMetaData
java.sql.ResultSetMetaData

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// This class provides information about the database as a whole.
//
// Many of the methods here return lists of information in ResultSets.
// You can use the normal ResultSet methods such as getString and getInt
// to retrieve the data from these ResultSets.  If a given form of
// metadata is not available, these methods many return a Java "null".
//
// Some of these methods take arguments that are String patterns.  These
// methods all have names such as fooPattern.  Within a pattern String "%"
// means match any substring of 0 or more characters and "_" means match
// any one character.
//

package java.sql;

public interface DatabaseMetaData {

    // First, a variety of minor information about the target database.

    // Can all the procedures returned by getProcedures be called by the
    // current user?
    boolean allProceduresAreCallable() throws SQLException;

    // Can all the table returned by getTable be SELECTed by the current user?
    boolean allTablesAreSelectable() throws SQLException;

    // If possible return the url for the database.
    // This should return java null if we can't generate a url.
    String getURL() throws SQLException;

    // get our user name as known to the database:
    String getUserName() throws SQLException;

    // is the connection in read-only mode?
    boolean isReadOnly() throws SQLException;

    // Some methods about how NULLs are sorted:
    boolean nullsAreSortedHigh() throws SQLException;
    boolean nullsAreSortedLow() throws SQLException;
    boolean nullsAreSortedAtStart() throws SQLException;
    boolean nullsAreSortedAtEnd() throws SQLException;

    // Get the name of the underlying database product.
    String getDatabaseProductName() throws SQLException;

    // get the database product version.
    String getDatabaseProductVersion() throws SQLException;

    // get the JDBC driver name
    String getDriverName() throws SQLException;

    // get the JDBC driver version as a String.
```

```
String getDriverVersion() throws SQLException;

// These methods return the JDBC major and minor versions of
// the JDBC driver supporting the connection.
int driverMajorVersion();
int driverMinorVersion();

// Does the database store each table in a local file?
boolean usesLocalFiles() throws SQLException;

// Does the database use a file for each table?
boolean usesLocalFilePerTable() throws SQLException;

// Does the database support mixed case unquoted SQL identifiers
// and how are the identifiers stored?
boolean supportsMixedCaseIdentifiers() throws SQLException;
boolean storedUpperCaseIdentifiers() throws SQLException;
boolean storesLowerCaseIdentifiers() throws SQLException;
boolean storesMixedCaseIdentifiers() throws SQLException;

// Does the database support mixed case quoted SQL identifiers
// and how are the quoted identifiers stored?
boolean supportsMixedCaseQuotedIdentifiers() throws SQLException;
boolean storedUpperCaseQuotedIdentifiers() throws SQLException;
boolean storesLowerCaseQuotedIdentifiers() throws SQLException;
boolean storesMixedCaseQuotedIdentifiers() throws SQLException;

// quote string used to surround quoted SQL identifiers?
String getIdentifierQuoteString() throws SQLException;

// get a comma separated list of all database specific SQL keywords.
String getSQLKeywords() throws SQLException;

// get a comma separated list of math functions.
String getNumericFunctions() throws SQLException;

// get a comma separated list of  string functions
String getStringFunctions() throws SQLException;

// get a comma separated list of system functions
String getSystemFunctions() throws SQLException;

// get a comma separated list of time and date functions
String getTimeDateFunctions() throws SQLException;

// This is the string that can be used to escape '_' or '%' in
// the string pattern based seacrhes such as getTables.
String getSearchStringEscape() throws SQLException;

// String of all the "extra" characters tha can be used in
// identifier names, i.e. those beyond a-z, 0-9 and _.
String getExtraNameCharacters() throws SQLException;

// Functions describing which features are supported.
```

```
// Do we support "ALTER TABLE" with add column?
boolean supportsAlterTableWithAddColumn() throws SQLException;

// Do we support "ALTER TABLE" with drop column?
boolean supportsAlterTableWithDropColumn() throws SQLException;

// Do we support column aliasing?
boolean supportsColumnAliasing() throws SQLException;

// Are concatenations between NUL and non-NUL values NULL?
boolean nullPlusNullIsNull() throws SQLException;

// Do we support the CONVERT function between SQL types?
boolean supportsConvert() throws SQLException;

// Do we support CONVERT betyween the given SQL types from java.sql.Types
boolean supportsConvert(int fromType, int toType)
                        throws SQLException;

// Do we support table correlation names?
boolean supportsTableCorrelationNames() throws SQLException;

// If we do support table correlation names, are they restricted to be
// differnet from the names of the tables?
boolean supportsDifferentTableCorrelationNames()
                        throws SQLException;

// Do we support expressions in "ORDER BY" lists?
boolean supportsExpressionsInOrderBy() throws SQLException;

// Can we use columns in "ORDER BY" not in the SELECT?
boolean suportsOrderByUnrelated() throws SQLException;

// Do we support some form of "GROUP BY" clause?
boolean supportsGroupBy() throws SQLException;

// Can a "GROUP BY" clause use columns not in the SELECT?
boolean supportsGroupByUnrelated() throws SQLException;

// Can a "GROUP BY" clauses add columns not in the SELECT?
boolean supportsGroupByBeyondSelect() throws SQLException;

// True if you cna use escape character sin "LIKE" clauses.
boolean supportsLikeEscapeClause() throws SQLException;

// Do we support multiple ResultSets from a single execute?
boolean supportsMultipleResultSets() throws SQLException;

// Can we have multiple transcations open at once (on different
// connections)?
boolean supportsMultipleTransactions() throws SQLException;

// Are there some columns that don't take NULL values?
```

```
boolean supportNonNullableColumns() throws SQLException;

// Sme methods about the level of SQL grammer supported
boolean supportsMinimumSQLGrammer() throws SQLException;
boolean supportsCodeSQLGrammer() throws SQLException;
boolean supportsExtendedSQLGrammer() throws SQLException;

// Extra SQL support for structual integrity?
boolean supportsIntegrityEnhancementFacility() throws SQLException;

// Some methods descibing outer-join support:
// Supports some form of outer join:
boolean supportsOuterJoins() throws SQLException;
// Full nested outer joins:
boolean supportsFullOuterJoins() throws SQLException;
// Outer joins with limits:
boolean supportsLimitedOuterJoins() throws SQLException;

// The database vendor's preferred term for "schema"
String getSchemaTerm() throws SQLException;

// The database vendor's preferred term for "procedure"
String getProcedureTerm() throws SQLException;

// The database vendor's preferred term for a "catalog"
String getCatalogTerm() throws SQLException;

// Does a catalog appear at the start of a auqlified table name?
// (Otherwise it appears at the end)
boolean isCatalogAtStart() throws SQLException;

// Separator between catalog and table name.
String catalogSeparator() throws SQLException;

// Kinds of statements where schema names can be used:
boolean supportsSchemasInDataManipulation() throws SQLException;
boolean supportsSchemasInProcedureCalls() throws SQLException;
boolean supportsSchemasInTableDefinitions() throws SQLException;
boolean supportsSchemasInIndexDefinitions() throws SQLException;
boolean supportsSchemasInPrivilegeDefinitions() throws SQLException;

// Kinds of statements where catalog names can be used:
boolean supportsCatalogsInDataManipulation() throws SQLException;
boolean supportsCatalogsInProcedureCalls() throws SQLException;
boolean supportsCatalogsInTableDefinitions() throws SQLException;
boolean supportsCatalogsInIndexDefinitions() throws SQLException;
boolean supportsCatalogsInPrivilegeDefinitions()
                        throws SQLException;

// Do we support positioned DELETE?
boolean supportsPositionedDelete() throws SQLException;

// Do we support positioned UPDATE?
boolean supportsPositionedUpdate() throws SQLException;
```

```
// Do we support SELECT for UPDATE?
boolean supportsSelectForUpdate() throws SQLException;

// Do we support stored procedures with a standard call syntax
boolean supportsStoredProcedures() throws SQLException;

// Situations in which subqueries are supported:
boolean supportsSubqueriesInComparisons() throws SQLException;
boolean supportsSubqueriesInExists() throws SQLException;
boolean supportsSubqueriesInIns() throws SQLException;
boolean supportsSubqueriesInQuantifieds() throws SQLException;

// are subqueries correlated?
boolean supportsCorrelatedSubqueries() throws SQLException;

// Support for SQL UNION:
boolean supportsUnion() throws SQLException;
boolean supportsUnionAll() throws SQLException;

// Normally all open statememts, resultsets, etc, are closed when
// either Statement.commit or Statement.rollback is called.
// However some databases provide mechanisms for preserving state
// across these methods.  The Connection.disableAutoCommit method
// can be used to try to keep statements open, and the following
// methods can be used to find what is supported:
boolean supportsOpenCursorsAcrossCommit();
boolean supportsOpenCursorsAcrossRollback();
boolean supportsOpenStatementsAcrossCommit();
boolean supportsOpenStatementsAcrossRollback();


// The following group of methods exposes various limitations
// based on the target database with the current driver.
// Unless othewise specified, a result of zero means there is no
// limit, or the limit is not known.

// How many hex characters can you have in an inline binary literal?
int getMaxBinaryLiteralLength() throws SQLException;

// What's the max length for a character literal?
int getMaxCharLiterallength() throws SQLException;

// What's the linit on column name length?
int getMaxColumnNameLength() throws SQLException;

// maximum numer of columns in a "GROUP BY" clause.
int getMaxColumnsInGroupBy() throws SQLException;

// maximum number of columns allowed in an index
int getMaxColumnsInIndex() throws SQLException;

// maximum number of columns in an "ORDER BY" clause
int getMaxColumnsInOrderBy() throws SQLException;
```

```
// maximum number of columns in a "SELECT" list
int getMaxColumnsInSelect() throws SQLException;

// maximum number of columns in a table
int getMaxColumnsInTable() throws SQLException;

// How many active connections can we have at a time to this database?
int getMaxConnections() throws SQLException;

// maximum cursor name length
int getMaxCursorNameLength() throws SQLException;

// maximum size of an index (in bytes)
int getMaxIndexLength() throws SQLException;

// The maximum size allowed for a schema name.
int getMaxSchemaNameLength() throws SQLException;

// The maximum length of a procedure name
int getMaxProcedureNameLength() throws SQLException;

// The maximum length of a catalog name.
int getmaxCatalogNameLength() throws SQLException;

// The maximum length of a single row
int getMaxRowSize() throws SQLException;

// And for the pernickity did getmaxRowSize() include LONGVARCHAR and
// LONGVARBINARY blobs?
boolean doesMaxRowSizeIncludeBlobs() throws SQLException;

// maximum length of a SQL statement
int getMaxStatementLength() throws SQLException;

// How many active statements can we have at a time to this database?
int getMaxStatements() throws SQLException;

// maximum length of a table name.
int getMaxtableNameLength() throws SQLException;

// maximum number of tables in a SELECT.
int maxTablesInSelect() throws SQLException;

// maximum length of a user name
int maxUserNameLength() throws SQLException;


// Get the databases's default transaction isolation level.
// The value are defined in java.sql.Connection.
int getDefaultTransactionIsolation() throws SQLException;

// if supportsTranscations returns false, then the databade doesn't
// support transactions.  This means that beginTransaction and commit
```

```
// are noops; and the isolation level is TRNSACTION_NONE.
boolean supportsTransactions() throws SQLException;

// Check if the database supports a givemn transaction isolation level.
// The values are defined in java.sql.Connection.
boolean supportsTransactionIsolationLevel(int level)
                        throws SQLException;

// Methdos specifying whether you can do data definition stamements as
// parts of transctions and what happens if you do.
boolean supportsDataDefinitionAndDataManipulationTransactions()
                         throws SQLException;
boolean supportsDataManipulationTransactionsOnly()
                        throws SQLException;
boolean dataDefinitionCausesTransactionCommit()
                        throws SQLException;
boolean dataDefinitionIgnoredInTransaction()
                        throws SQLException;


// getProcedures returns a ResultSet that contains a row for each matching
// stored procedure in the database.
// Each row in the ResultSet contains the following fields:
//  Column(1) String => procedure catalog (may be NULL)
//  Column(2) String => procedure schema (may be NULL)
//  Column(3) String => procedure name
//  Column(7) String => explanatory comment on the procedure
//  Column(8) short  => kind of procedure:
int procedureResultUnknown= 0;// May returns a result.
int procedureNoResult= 1;// Does not return a result
int procedureReturnsResult= 2;// Returns a result.
// The output can be limited by specifying catalog, schema, and procedure
// name patterns that must be matched.
// The results are sorted by columns 1, 2, and 3.

ResultSet getProcedures(String catalog, String schemaPattern,
        String procedureNamePattern) throws SQLException;

// getProcedureColumns returns a ResultSet that contains rows describing
// the parameters and result columns from matching stored procedures in
// the database.
// Each row in the ResultSet contains the following fields:
//  Column(1)  String => procedure catalog (may be NULL)
//  Column(2)  String => procedure schema (may be NULL)
//  Column(3)  String => procedure name
//  Column(4)  String => column/parameter name
//  Column(5)  Short  => kind of column/parameter:
int procedureColumnUnknown = 0;// nobody knows
int procedureColumnIn= 1;// INT parameter
int procedureColumnInOut= 2;// INOUT parameter
int procedureColumnResult= 3;// result column in ResultSet
int procedureColumnOut= 4;// OUT parameter
int procedureColumnReturn= 5;// return value of function
//  Column(6)  short  => SQL type from java.sql.Types
```

```
//  Column(7)  String => SQL type name
//  Column(8)  int    => precision
//  Column(10) short  => scale
//  Column(11) short  => radix
//  Column(12) boolean => can it contain null?
//  Column(13) String => comment describing parameter/column
// The results are sorted by columns 1, 2, 3, and 5.

ResultSet getProcedureColumns(String catalog,
        String schemaPattern,
        String procedureNamePattern,
        String columnNamePattern) throws SQLException;

// getTables returns a ResultSet that contains a row for each matching
// table in the database
// Each row in the ResultSet contains the following fields:
//  Column(1) String => table catalog (may be NULL)
//  Column(2) String => table schema
//  Column(3) String => table name
//  Column(4) String => table type.  Typical types are "TABLE",
//          "VIEW","SYSTEM TABLE", "GLOBAL TEMPORARY",
//          "LOCAL TEMPORARY", "ALIAS", "SYNONYM".
//  Column(5) String => explanatory comment on the table
// The output can be limited by specifying catalog, schema, and procedure
// name patterns that must be matched.  If the table types array is non-null
// then only tables of the given types will be returned.
// The results are sorted by columns 4, 1, 2, and 3.

ResultSet getTables(String catalog, String schemaPattern,
    String tableNamePattern, String types[]) throws SQLException;

// getSchemas returns a ResultSet containing a row for each schema.
// Each row in the ResultSet contains a single String field which is the
// schema username.
// The results are sorted by string value.

ResultSet getSchemas() throws SQLException;

// getCatalogs returns a ResultSet containing a row for each "catalog" in
// the database. Each row in the ResultSet contains a single String field
// which is the catalog name.
// The results are sorted by string value.

ResultSet getCatalogs() throws SQLException;

// getTableTypes returns a ResultSet containing a row for each table type in
// the database. Each row in the ResultSet contains a single String field which
// is the table type.
// The results are sorted by string value.

ResultSet getTableTypes() throws SQLException;

// getColumns returns a ResultSet containing a row for each column in the
// matching table(s).
```

```
// The ResultSet is order by catalog, schema, and table name.
// Each row in the ResultSet contains:
//  Column(1)  String => table catalog (may be NULL)
//  Column(2)  String => table schema
//  Column(3)  String => table name
//  Column(4)  String => column name
//  Column(5)  short   => SQL type from java.sql.Types
//  Column(6)  String => SQL type name
//  Column(7)  int  => column size.  For char or date types this
//               is the maximum number of characters, for
//               numeric or decimal types this is precision.
//  Column(8)  is not used.
//  Column(9)  int  => scale
//  Column(10) int  => Radix (typicaly either 10 or 2)
//  Column(11) boolean=> can column contain NULL?
//  Column(12) String => comment describing column (may be NULL)
//  Column(13) String => default value (may be NULL)
//  Column(14) is not used
//  Column(15) int    => for char types the maximum number of bytes
//               in the column.  May be NULL.
//  Column(16) int  => index of column in table (starting at 1)
//               may be NULL.
//  Column(17) String => "NO" is column is know not to be nullable,
//               "YES" otherwise.  May be NULL.
//
// The results are sorted by columns 1, 2, and 3.

ResultSet getColumns(String catalog, String schemaPattern,
    String tableNamePattern, String columnNamePattern)
                throws SQLException;

// getColumnPrivileges returns a ResultSet containing a row for each set
// of access rights which matches the given pattern for column names.
// Each row in the ResultSet contains:
//  Column(1) String => table catalog (may be NULL)
//  Column(2) String => table schema
//  Column(3) String => table name
//  Column(4) String => column name
//  Column(5) String => grantor of access
//  Column(6) String => grantee of access
//  Column(7) String => name of access (SELECT, INSERT, UPDATE, REFRENCES, ...)
//  Column(8) String => grantable ("YES" if it can be granted to others)
// The results are sorted by columns 1, 2, 3, 4, and 7.

ResultSet getColumnPrivileges(String catalog, String schema,
    String table, String columnNamePattern) throws SQLException;

// getTablePrivileges returns a ResulSet containing a row for each set of
// access rights associated with matching tables.
// Each row in the ResultSet contains:
//  Column(1) String => table catalog (may be NULL)
//  Column(2) String => table schema
//  Column(3) String => table name
//  Column(4) String => grantor of access
```

```
//  Column(5) String => grantee of access
//  Column(6) String => name of access (SELECT, INSERT, UPDATE, REFRENCES, ...)
//  Column(7) String => grantable ("YE"S if it can be granted to others)
// The results are sorted by columns 1, 2, 3, and 7.

ResultSet getTablePrivileges(String catalog, String schemaPattern,
            String tableNamePattern) throws SQLException;

// getBestRowIdentifier returns the optimal set of columns that can be
// used to select rows in the given table.
// The input and output scope values are one of
int bestRowTemporary   = 0; // => very temporary, while using row
int bestRowTransaction = 1; // => valid for remainder of current transaction
int bestRowSession     = 2; // => valid for remainder of current session
// The "nullable" argument specifies whether to return columns that can
// take null values.
// Each row in the ResultSet contains:
//  Column(1) short  => actual scope of result (see above)
//  Column(2) String => column name
//  Column(3) short  => SQL data type from java.sql.Types
//  Column(4) String => SQL type name
//  Column(5) int    => precision
//  Column(7) short => scale
//  Column(8) short  => is this a psuedo column:
int bestRowUnknown= 0;
int bestRowNotPsuedo= 1;
int bestRowPseudo= 2;
// The results are sorted by column 1.

ResultSet getBestRowIdentifier(String catalog, String schema,
    String table, int scope, boolean nullable) throws SQLException;

// getVersionColumns returns the set of column names where the column
// is automaticaly updated when ever any field in a row changes.
// Each row in the ResultSet contains:
//  Column(1) is not used
//  Column(2) String => column name
//  Column(3) short  => SQL data type from java.sql.Types
//  Column(4) String => SQL type name
//  Column(5) int    => precision
//  Column(7) short => scale
//  Column(8) short  => is this a psuedo column?
int versionColumnUnknown= 0;
int versionColumnNotPseudo= 1;
int versionColumnPseudo= 2;
// The results are unsorted.

ResultSet getVersionColumns(String catalog, String schema,
            String table) throws SQLException;

// getPrimaryKeys returns infomation about the primary keys for
// the given table.
// Each row in the ResultSet contains:
//  Column(1) String => table catalog (may be NULL)
```

```
//  Column(2) String => table schema
//  Column(3) String => table name
//  Column(4) String => column name
//  Column(5) short  => sequence number within primary key
//  Column(6) String => primary key name
// The results are sorted by columns 1, 2, 3, and 5.

ResultSet getPrimaryKeys(String catalog, String schema,
            String table) throws SQLException;

// getImportedKeys returns information about the foreign keys in the
// given table, and the primary keys to which they refer.  See also
// getExportedKeys and getCrossReference.
//
// Each row in the ResultSet contains:
//  Column(1)  String => primary key table catalog
//  Column(2)  String => primary key table schema
//  Column(3)  String => primary key table name
//  Column(4)  String => primary key column name
//  Column(5)  String => foreign key table catalog
//  Column(6)  String => foreign key table schema
//  Column(7)  String => foreign key table name
//  Column(8)  String => foreign key table name
//  Column(9)  String => column sequence number within primary key
//  Column(10) short  => Update rule.  What happens to foreign key
//               when primary is updated (may be NULL):
int importedKeyCascade= 0;
int importedKeyRestrict = 1;
int importedKeySetNull  = 2;
//  Column(11) short  => Delete rule.  What happens to the foreign
//               key when primary is deleted.  Values
//               are the same as for the update rule.
//  Column(12) String => foreign key name (may be NULL)
//  Column(13) String => primary key name (may be NULL)
// The results are sorted by columns 1, 2, 3, and 9.

ResultSet getImportedKeys(String catalog, String schema,
            String table) throws SQLException;

// getExportedKeys returns information about foreign keys in other
// tables which reference the primary key in this table.  See also
// getImportedKeys and getCrossReference.
//
// The columns in the ResultSet ares as for getImportedKeys.
// The results are sorted by columns 5, 6, 7, and 9.

ResultSet getExportedKeys(String catalog, String schema,
            String table) throws SQLException;

// getCrossReference returns information about the foreign key in
// foreignTable which references the primary key for primaryTable.
// See also getExportedKeys and getImportedKeys.
//
// The columns in the ResultSet ares as for getImportedKeys.
```

```
// The results should be either zero or one rows and are not sorted.

ResultSet getCrossReference(
    String primaryCatalog, String primarySchema, String primaryTable,
    String foreignCatalog, String foreignSchema, String foreignTable
    ) throws SQLException;

// getTypeInfo returns information on all the standard SQL types
// supported by the target database.
//
// The columns in the ResultSet are:
//  Column (1)  String => Type name
//  Column (2)  int    => SQL data type from java.sql.Types
//  Column (3)  int    => maximum precision
//  Column (4)  String => prefix used to quote a literal
//  Column (5)  String => suffix used to quote a literal
//  Column (6)  String => parameters used in creating the type
//  Column (7)  boolean=> can you use NULL for this type?
//  Column (8)  boolean=> is it case sensitive?
//  Column (9)  boolean=> can you use "WHERE" based on this type:
int typeUnSearchable   = 0;// No support
int typeSearchLikeOnly = 1;// Only supported with WHERE .. LIKE
int typeSearchNotLike  = 2;// Supported except for WHERE .. LIKE
int typeSearchable     = 3;// Supported for all WHERE ..
//  Column (10) boolean=> is it unsigned?
//  Column (11) boolean=> can it be a money value?
//  Column (12) boolean=> can it be used for an auto-increment value?
//  Column (13) String => localized version of type name
//  Column (14) short  => minimum scale supported
//  Column (15) short  => maximum scale supported
//
// The results are sorted by columns 2 and 1.

ResultSet getTypeInfo() throws SQLException;

// getIndexInfo returns information about indexes for a given table.
// If the "unique" argument is true then the results describes only
// the indexes that are restricted to unique values.  If the
// "approximate" argument is true then the driver may use cached
// and protentially out of date values for some ResultSet fields.
//
// The columns in the ResultSet are:
//  Column (1)  String => catalog (may be NULL)
//  Column (2)  String => schema (may be NULL)
//  Column (3)  String => table name
//  Column (4)  short  => Is this index always unique?
//  Column (5)  String => index catalog (may be NULL)
//  Column (6)  String => index name
//  Column (7)  short  => index type:
short tableIndexStatistic = 0;
short tableIndexClustered = 1;
short tableIndexHashed    = 2;
short tableIndexOther     = 3;
//  Column (8)  short  => column sequence number within index
```

```
        //  Column (9)  String => column name
        //  Column (10) String => column sort sequence, "A" => ascending,
        //              "D" => descending, may be NULL.
        //  Column (11) int  => If type == tableIndexStatisic then this
        //           is the number of rows in the table, otherwise
        //              it is the number of unique valies in the index.
        //           may be NULL.
        //  Column (12) String => If type == tableIndexStatisic then this
        //           is the number fo pages used for the table,
        //               otherwise it is the mnumber of pages used for
        //           the current index.  May be NULL.
        //  Column (13) String => Filter condition, if any.  (may be NULL)
        //
        // The results are sorted by columns 4, 5, 6, and 8.

        ResultSet getIndexInfo(String catalog, String schema, String table,
                boolean unique, boolean approximate)
                        throws SQLException;

    }
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// A ResultMetaData object can be used to find out about the types
// and properties of the columns in a ResultSet.

package java.sql;

public interface ResultSetMetaData {

    // The number of columns in the ResultSet.
    int getColumnCount() throws SQLException;

    // Is column utomaticallty numbered, thus read-only.
    boolean isAutoIncrement(int column) throws SQLException;

    // Does case matter?
    boolean isCaseSensitive(int column) throws SQLException;

    // Can column be used in where clause?
    boolean isSearchable(int column) throws SQLException;

    // Is it a cash value?
    boolean isCurrency(int column) throws SQLException;

    // Can you put a null in this column?
    boolean isNullable(int column) throws SQLException;

    // True if column is signed number.
    boolean isSigned(int column) throws SQLException;

    // normal max width as chars
    int getColumnDisplaySize(int column) throws SQLException;

    // Suggested column title for use in printouts and displays
    String getColumnLabel(int column) throws SQLException;

    // SQL name for column
    String getColumnName(int column) throws SQLException;

    // Column's table's schema, or "" if not applicable.
    String getSchemaName(int column) throws SQLException;

    // Number of decimal digits
    int getPrecision(int column) throws SQLException;

    // Number of digits to right of decimal
    int getScale(int column) throws SQLException;

    // name of column's table, or "" if not applicable.
    String getTableName(int column) throws SQLException;

    // catalog of column's table, or "" if not applicable.
    String getCatalogName(int column) throws SQLException;
```

```
    // SQL type, from java.sql.Types
    int getColumnType(int column) throws SQLException;

    // SQL type name
    String getColumnTypeName(int column) throws SQLException;

    // definitely not writable
    boolean isReadOnly(int column) throws SQLException;

    // a write *may* succeed
    boolean isWritable(int column) throws SQLException;

    // a write will succeed
    boolean isDefinitelyWritable(int column) throws SQLException;
}
```

```
// Copyright (c) 1995 Sun Microsystems, Inc. All Rights Reserved.

// A ParameterMetaData object provides information about the parameters
// to a PrepareStatement.

package java.sql;

public interface ParameterMetaData {

    // How many parameters are there?
    int getParameterCount() throws SQLException;

    // Information on individual parameters:

    // SQL type id from java.sql.Types
    int getParameterType(int index) throws SQLException;

    // Number of decimal digits
    int getPrecision(int column) throws SQLException;

    // Number of digits to right of decimal
    int getScale(int column) throws SQLException;

    // Can you put a null for this parameter?
    boolean isNullable(int column) throws SQLException;
}
```

Appendix A: Rejected design choices

A.1 Use Holder types rather than get/set methods.

In earlier drafts of JDBC we used a mechanism of "Holder" types to pass parameters and to obtain results. This mechanism was an attempt to provide a close analogue to the use of pointers to variables in ODBC. However as we tried to write test examples we found the need to create and bind Holder types fairly irksome, particularly when processing simple row results.

We therefore came up with the alternative design using the setParameter and "get" methods that is described in Sections 7.3 and 7.2. After comparing various example programs we decided that the setParameter/get mechanism seemed to be simpler for programmers to use. It also removed the need to define a dozen of so Holder types as part of the JDBC API. So we are currently proposing to use the setParameter/get mechanism and not to use Holders.

A.1.1 Using Holder types to pass parameters

As part of the java.sql API we define a set of Holder types to hold parameters to SQL statements. There is an abstract base class Holder, and then specific subtypes for different Java types that may be used with SQL. For example there is a StringHolder to hold a String parameter and a ByteHolder to hold a byte parameter.

To allow parameters to be passed to SQL statements, the java.sql.Statement class allows you to associate Holder objects with particular parameters. When the statement is executed any IN or INOUT parameter values will be read from the corresponding Holder objects and when the statement completes then any OUT or INOUT parameters will get written back to the corresponding Holder objects.

An example of IN parameters using Holders:

```
java.sql.Statement stmt = conn.createStatement();
// We pass two parameters. One varies each time around
// the for loop, the other remains constant.
IntHolder ih = new IntHolder();
stmt.bindParameter(1, ih);
StringHolder sh = new StringHolder();
stmt.bindParameter(2, sh);
sh.value ="Hi"
for (int i = 0; i < 10; i++) {
       ih.value = i;
       stmt.execute("UPDATE Table2 set a = ? WHERE b = ?");
}
```

An example of OUT parameters using Holders:

```
java.sql.Statement stmt = conn.createStatement();
IntHolder ih = new IntHolder();
stmt.bindParameter(1, ih);
StringHolder sh = new StringHolder();
stmt.bindParameter(2, sh);
for (int i = 0; i < 10; i++) {
       stmt.execute("{CALL testProcedure(?, ?)}");
       byte x = ih.value;
       String s = sh.value;
}
```

A.1.2 Getting row results using Holder objects

Before executing a statement we allow the application programmers to bind Holder objects to particular columns. After the statement has executed, the application program can iterate over the ResultSet using ResultSet.next() to move to successive rows. As the application moves to each row, the Holder objects will be populated with the values in that row. This is similar to the SQLBindColumn mechanism used in ODBC.

Here's a simple example:

```
// We're going to execute a SQL statement that will return a
// collection of rows, with column 1 as an int, column 2 as
// a String, and column 3 as an array of bytes.
java.sql.Statement stmt = conn.createStatement();
IntHolder ih = new IntHolder();
stmt.bindHolder(1, ih);
StringHolder sh = new StringHolder();
stmt.bindHolder(2, sh);
BytesHolder bh = new BytesHolder();
stmt.bindHolder(3, bh);
ResultSet r = stmt.executeQuery("SELECT a, b, c FROM Table7");
while (r.next()) {
       // print the values for the current row.
       int i = ih.value;
       String s = sh.value;
       byte b[] = bh.value;
       System.out.println("ROW = " + i + " " + s + " " + b[0]);
}
```

A.2 Design Alternative: Don't use types such as fooHolder, instead use foo[]

At some point in the future we would probably like to add support for some form of column-wise binding, so that a bunch of rows can be read at once. When we were using the Holder design, we considered the following design alternative that would allow for column-wise binding.

Holder objects are capable of holding single instances of various Java types. However an array of a single element could instead be used as a holder.

This approach has several disadvantages, but one major advantage.

The first disadvantage is that people may be confused if they read "foo f[] = new foo[1];". The corresponding holder declaration "fooHolder f = new fooHolder();" gives a better clue as to what f is and why we are allocating it.

The second disadvantage is that we would have to replace the single methodStatement.bindColumn with a distinct method for each array type. This is because all our Holder types inherit from java.sql.Holder and can therefore be passed as arguments to a generic method that takes a java.sql.Holder argument. (On the other hand at least we avoid defining the dozen or so holder classes.)

The last disadvantage is that using foo[] only gives us the raw java type information. By defining a specific set of holder types for use with SQL, we can define extra fields and/or semantics, e.g. for the CurrencyHolder type.

The corresponding major advantage is that if we use foo[1] as the container for a parameter then it is very natural to allow foo[x] as a way of binding multiple rows of a table in column-wise binding. This would let us add support for column-wise binding without having to remodel the interface.

If we use arrays instead of Holders, them the bindColumn mechanism makes it easier to scale up to column-wise binding.

A.3 Support for retrieving multiple rows at once

Currently we provide methods for retrieving individual columns within individual rows, a field at a time. We anticipate that drivers will normally prefetch rows in larger chunks so as to reduce the number of interactions with the target database. However it might also be useful to allow programmers to retrieve data in larger chunks through the JDBC API.

The easiest mechanism to support in Java would probably be to support some form of column-wise binding where a programmer can specify a set of arrays to hold (say) the next 20 values in each of the columns, and then read all 20 rows at once.

However we do not propose to provide such a mechanism in the first version of JDBC. We do recommend that drivers should normally prefetch rows in suitable chunks.

Appendix B: Example JDBC Programs

B.1 Using SELECT

```
import java.net.URL;
import java.sql.*;

class Select {

     public static void main(String argv[]) {
        try {
            // Create a URL specifying an ODBC data source name.
            String url = "jdbc:odbc:wombat";

            // Connect to the database at that URL.
            Connection con = Environment.getConnection(url, "kgh", "");

            // Execute a SELECT statement
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT a, b, c, d, key FROM Table1");

            // Step through the result rows.
            System.out.println("Got results:");
            while (rs.next()) {
                // get the values from the current row:
                int a = rs.getInteger(1);
                Numeric b = rs.getNumeric(2);
                char c[] = rs.getVarChar(3).tocharArray();
                boolean d = rs.getBit(4);
                String key = rs.getVarChar(5);

                // Now print out the results:
                System.out.print("  key=" + key);
                System.out.print("  a=" + a);
                System.out.print("  b=" + b);
                System.out.print("  c=");
                for (int i = 0; i < c.length; i++) {
                    System.out.print(c[i]);
                }
                System.out.print("  d=" + d);
                System.out.print("\n");
            }

            stmt.close();
            con.close();
        } catch (java.lang.Exception ex) {
            ex.printStackTrace();
        }
     }
}
```

B.2 Using UPDATE

```
// Update a couple of rows in a database.

import java.net.URL;
import java.sql.*;

class Update {

    public static void main(String argv[]) {
        try {
            // Create a URL specifying an ODBC data source name.
            String url = "jdbc:odbc:wombat";

            // Connect to the database at that URL.
            Connection con = Environment.getConnection(url, "kgh", "");

            // Create a prepared statement to update the "a" field of a
            // row in the "Table1" table.
            // The prepared statement takes two parameters.
            PreparedStatement stmt = con.prepareStatement(
                        "UPDATE Table1 SET a = ? WHERE key = ?");

            // First use the prepared statement to update the "count" row.
            // the "count" row to 34.
            stmt.setInteger(1, 34);
            stmt.setVarChar(2, "count");
            stmt.execute();
            System.out.println("Updated \"count\" row OK.");

            // Now use the same prepared statement to update the
            // "mirror" field.
            // We rebind parameter 2, but reuse the other parameter.
            stmt.setVarChar(2, "mirror");
            stmt.execute();
            System.out.println("Updated \"mirror\" row OK.");

            stmt.close();
            con.close();

        } catch (java.lang.Exception ex) {
            ex.printStackTrace();
        }
    }
}
```