sun microsystems

JavaSoft

JDBC ™: A Java SQL API

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

Please send technical comments on this specification to:

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.

Please send product and business questions to:

jdbc-business@wombat.eng.sun.com

Graham Hamilton & Rick Cattell

Version 1.00

June 12, 1996

Copyright ©1996 by Sun Microsystems Inc.

2550 Garcia Avenue, Mountain View, CA 94043.

All rights reserved.

RESTRICTED RIGHTS: Use, duplication or disclosure by the government is subject to the restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause as DFARS 252.227-7013 and FAR 52.227-19.

Sun, Sun Microsystems, the Sun logo, Java, JavaSoft, JDBC, and JDBC COMPLIANT are trademarks or registered trademarks of Sun Microsystems, Inc.

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MECHANTABILITY, FITNESS FOR A PARTICULAR USE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC., MAY MAKE NEW IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

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.

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 decided 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.

Our immediate priority has been to define a common low-level API that supports basic SQL functionality. We call this API JDBC. This API in turn allows the development of higher-level database access tools and APIs.

Fortunately we didn't need to design a SQL API from scratch. We based our work on the X/Open SQL CLI (Call Level Interface) which is also the basis for Microsoft's ODBC interface. Our main task has been 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 thank the many reviewers in the database, database connectivity, and database tools communities who gave generously of their time and expertise.

We are grateful to all the reviewers who took the time to read and comment on the various drafts, but we would particularly like to thank Azad Bolour, Paul Cotton, Ed Garon, John Goodson, Mark Hapner, Tommy Hawkins, Karl Moss, and Barbara Walters for providing sustained review and helpful feedback over many drafts and revisions of the API.

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

JDBC is intended as 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 SQL92 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 parameters and large blobs.

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.

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. Metadata functions are therefore documented at the end of the specification, along with dynamically-typed data access; the average programmer can skip these sections.

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.

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 method 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.

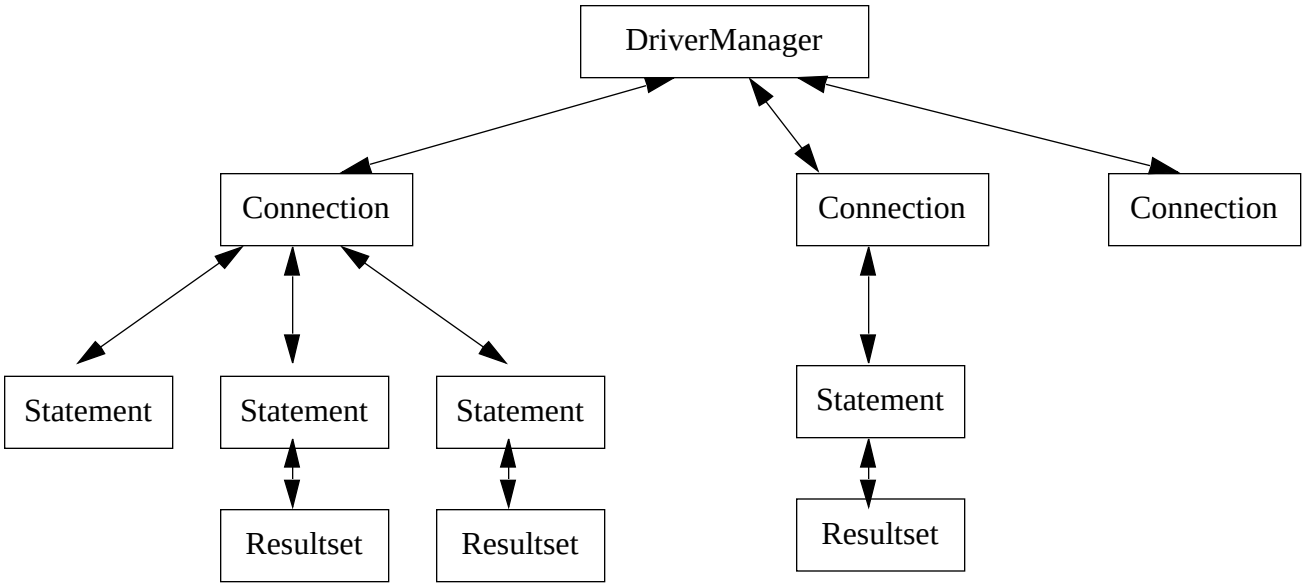
We've tried to adopt the same approach for the JDBC interface, and in general have preferred to use multiple methods rather than using multi-purpose methods with flag arguments.

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.

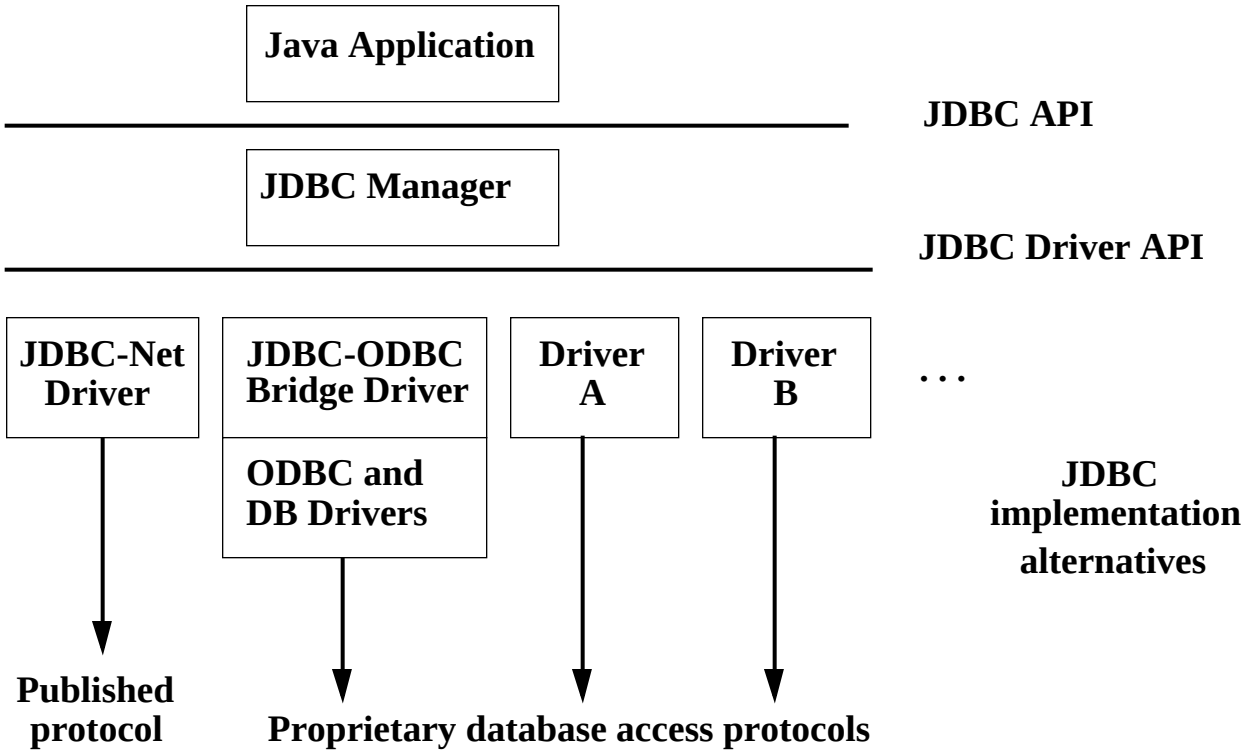


The most important interfaces are:

- java.sql.DriverManager which handles loading of drivers and 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

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. In addition the JDBC interfaces for discovering database metadata are described in Chapter 15.



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.DriverManager class when it needs to locate a driver for a particular database URL.

JavaSoft is providing 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.

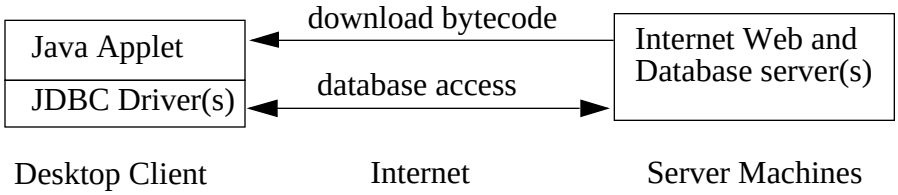
Another useful driver is one that goes directly to a DBMS-independent network protocol. It would be desirable to publish the protocol to allow multiple server implementations, e.g. on top of ODBC or on specific DBMSs (although there are already products that use a fixed protocol such as this, we are 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 itself is very small and efficient as well. The net result is a very small and fast all-Java client side implementation that speaks to any server speaking the published protocol.

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:

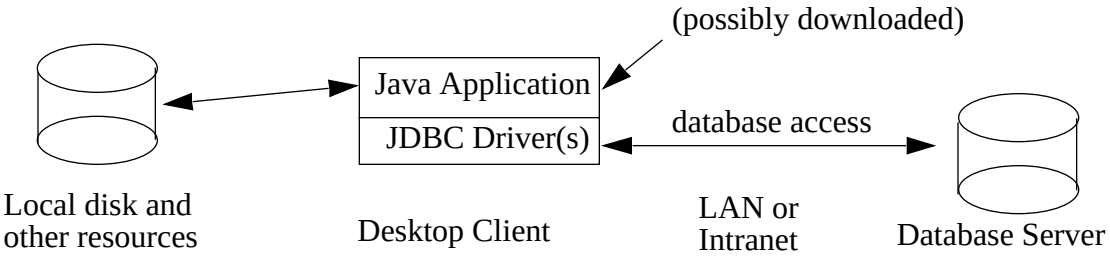
- Untrusted applets are severely constrained in the operations they are allowed to perform. In particular, they are not allowed any access to local files and they are not allowed to open network connections to arbitrary hosts.
- Applets on the Internet present new problems with respect to identifying 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.

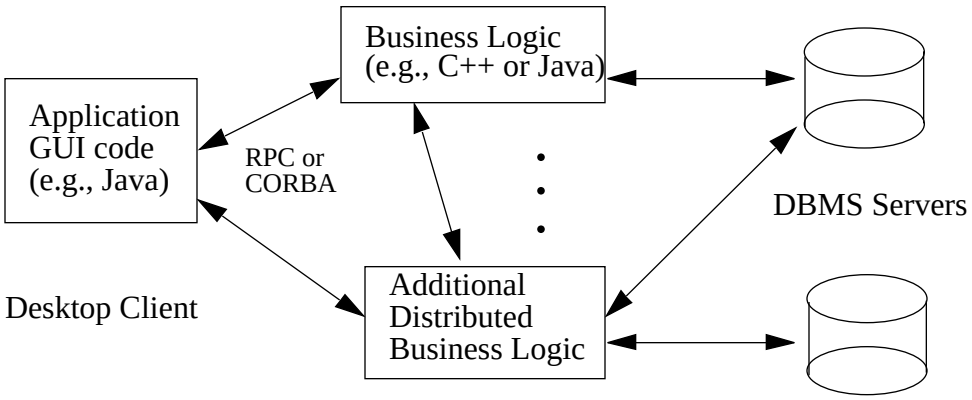
The Java application and "Intranet" cases differ from applets in a number of ways. For example, the most natural way to identify a database is typically for the user or application to specify a database name, e.g. "Customers" or "Personnel". The users will expect the system to locate the specific machine,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

The security of database requests and data transmission on the network, especially in the Internet case, is also an important consideration for the JDBC user. However, keep in mind that we are defining programming interfaces in this specification, not a network protocol. The network protocols used for database accessed have generally already been fixed by the DBMS vendor or connectivity vendor. A JDBC user should verify that the network protocol provides adequate security for their needs before using a JDBC driver and DBMS server.

If JavaSoft proposes a standard for a published protocol for a DBMS-independent JDBC-Net driver as described in Section 3, then security considerations will be an important factor in the selection of protocol.

5.4 Security Responsibilities of Drivers

Because JDBC drivers may be used in a variety of different situations, it is important that driver writers follow certain simple security rules to prevent applets from making illegal database connections.

These rules are unnecessary if a driver is downloaded as an applet, because the standard security manager will prevent an applet driver from making illegal connections. However JDBC driver writers should bear in mind that if their driver is "successful" then users may start installing it on their local disks, in which case it becomes a trusted part of the Java environment, and must make sure it is not abused by visiting applets. We therefore urge all JDB driver writers to follow the basic security rules.

These rules all apply at connection open time. This is the point when the driver and the virtual machine should check that the current caller is really allowed to connect to a given database. After connection open, no additional checks are necessary.

5.4.1 Be very careful about sharing TCP connections

If a JDBC driver attempts to open a TCP connection then the open will be automatically checked by the Java security manager. The security manager will check if there is an applet on the current call stack and if so, will restrict the open to whatever set of machines that applet is allowed to call. So normally a JDBC driver can leave TCP open checks up to the Java virtual machine.

However if a JDBC driver wants to share a single TCP connection between several different database connections then it becomes the driver's responsibility to make sure that each of its callers is really allowed to talk to the target database. For example, if we open a TCP connection to the machine foobah for applet A, this does not mean that we should automatically allow applet B to share that connection. Applet B may have no right whatsoever to access machine foobah.

So before allowing someone to re-use an existing TCP connection the JDBC driver should check with the security manager that the current caller is allowed to connect to that machine. This can be done with the following code fragment:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkConnect(hostName, portNumber);
}
```

The Security.checkConnect method will throw a java.lang.SecurityException if the connection is not permitted.

5.4.2 Check all local file access

If a JDBC driver needs to access any local data on the current machine then it must ensure that its caller is allowed to open the target files. For example

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkRead(fileName);
}
```

The Security.checkRead method will throw a java.lang.SecurityException if the current caller is an applet which is not allowed to access the given file.

As with TCP connections, the driver need only be concerned with these security issues if file resources are shared among multiple calling threads and the driver is running as trusted code.

5.4.3 Assume the worst

Some drivers may use native methods to bridge to lower level database libraries. In these cases it may be difficult to determine what files or network connections will be opened by the lower level libraries.

In these circumstances the driver must make “worse case” security assumptions and deny all database access to downloaded applets unless the driver is completely confident that the intended access is innocuous.

For example a JDBC-ODBC bridge might check the meaning of ODBC data source names and only allow an applet to use those ODBC data source names that reference databases on machines to which the applet is allowed to open connections (see 5.4.1 above). But for some ODBC data source names the driver may be unable to determine the hostname of the target database and must therefore deny downloaded applets access to these data sources.

In order to determine if the current caller is a trusted application or applet (and can therefore be allowed arbitrary database access) the JDBC driver can check if the caller is allowed to write an arbitrary file:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkWrite(“foobaz”);
}
```

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.DriverManager.getConnection method.

The DriverManager.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) if it can connect to the given URL.[1] Drivers 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, java.sql.PreparedStatement, and java.sql.CallableStatement 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 colon 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.

1. At first glance this may seem inefficient, but keep in mind that this requires only a few procedure calls and string comparisons per connection since it is unlikely that dozens of drivers will concurrently be loaded.

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.

6.3.6 The “odbc” sub-protocol

The “odbc” sub-protocol has been reserved for URLs that specify ODBC style Data Source Names. For this sub-protocol we specify a URL syntax that allows arbitrary attribute values to be specified after the data source name.

The full odbc sub-protocol URL syntax is:

```
jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]*
```

Thus valid jdbc:odbc names include:

```
jdbc:odbc:qeor7
jdbc:odbc:wombat
jdbc:odbc:wombat;CacheSize=20;ExtensionCase=LOWER
jdbc:odbc:qeora;UID=kgh;PWD=fooey
```

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.DriverManager class initializes it will look for a “sql.drivers” property in the system properties. If the property exists it should consist of a colon-separated list of driver class names. Each of the named classes should implement the java.sql.Driver interface. The DriverManager class will attempt to load each named Driver class.

Second, a programmer can explicitly load a driver class using the standard Class.forName method. For example, to load the acme.db.Driver class you might do:

```
Class.forName(“acme.db.Driver”);
```

In both cases it is the responsibility of each newly loaded Driver class to register itself with the DriverManager, using the DriverManager.registerDriver method. This will allow the DriverManager to use the driver when it is attempting to make database connections.

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 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.getInt("a");
      String s = r.getString("b");
      byte b[] = r.getBytes("c");
      System.out.println("ROW = " + i + " " + s + " " + b[0]);
}
```

There are two alternative ways of naming columns. You can either use column indexes (for greater efficiency) or column names (for greater convenience). Thus for example there is both a getString method that takes a column index and a getString method that takes a column name.

Reviewer input convinced us that we had to support both column indexes and column names. Some reviewers were extremely emphatic that they require highly efficient database access and therefore preferred column indexes, other reviewers insisted that they wanted the convenience of using column names. (Note that certain SQL queries can return tables without column names or with multiple identical column names. In these cases, programmers should use column numbers.)

For maximum portability, columns within a row should be read in left-to-right order and each column should only be read once. This reflects implementation limitations in some underlying database protocols.

7.1.1 Data conversions on query results

The ResultSet.getXXX methods will attempt to convert whatever SQL type was returned by the database to whatever Java type is returned by the getXXX method.

Table 1 on page 21 lists the supported conversions from SQL types to Java types via getXXX methods. For example, it is possible to attempt to read a SQL VARCHAR value as an integer using getInt, but it is not possible to read a SQL FLOAT as a java.sql.Date.

If you attempt an illegal conversion or if a data conversion fails (for example if you did a getInt on a SQL VARCHAR value of "foo") then a SQLException will be raised.

	TYINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getByte	**X**	x	x	x	x	x	x	x	x	x	x	x	x						
getShort	x	**X**	x	x	x	x	x	x	x	x	x	x	x						
getInt	x	x	**X**	x	x	x	x	x	x	x	x	x	x						
getLong	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						
getDouble	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						
getBoolean	x	x	x	x	x	x	x	x	x	**X**	x	x	x						
getString	x	x	x	x	x	x	x	x	x	x	**X**	**X**	x	x	x	x	x	x	x
getBytes														**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 1: Use of ResultSet.getXXX methods to retrieve common SQL data types
An “x” means that the given getXXX method can be used to retrieve the given SQL type.
An “**X**” means that the given getXXX method is recommended for retrieving the given SQL type

7.1.2 Null result values

To determine if a given result value is SQL “NULL” you must first read the column and then use the ResultSet.wasNull method to discover if the read returned a SQL “NULL”. (See also Appendix A.9).

When you read a SQL “NULL” using one of the ResultSet.getXXX methods, you will receive:

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

7.1.3 Retrieving very large row values.

JDBC allows arbitrarily large LONGVARBINARY or LONGVARCHAR data to be retrieved using getBytes and getString, 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.**

Java streams return untyped bytes and can (for example) be used for both ASCII and Unicode. We define three separate methods for getting streams. GetBinaryStream returns a stream which simply provides the raw bytes from the database without any conversion. GetAsciiStream returns a stream which provides one byte ASCII characters. GetUnicodeStream returns a stream which provides 2 byte Unicode characters.

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(“x”);
       for (;;) {
             int size = fin.read(buff);
             if (size == 0) {
                   break;
             }
             // Send the newly filled buffer to some ASCII output stream:
             output.write(buff, 0, size);
       }
}
```

7.1.4 Optional or multiple ResultSets

Normally we expect that SQL statements will be executed using either executeQuery (which returns a single ResultSet) or executeUpdate (which can be used for any kind of database modification statement and which returns a count of the rows updated).

However 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 and/or update counts.

To accommodate these needs we provide a mechanism so that an application can execute a statement and then process an arbitrary collection of ResultSets and update counts. This mechanism is based on a fully general “execute” method, supported by three other methods, getResultSet, getUpdateCount, and getMoreResults (see Section 13). These methods allow an application to explore the statement results one at a time and to determine if a given result was a ResultSet or an update count.

7.2 Passing IN parameters

To allow you to pass parameters to a SQL statement, the java.sql.PreparedStatement class provides a series of setXXX 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.setString(1, “Hi”);
for (int i = 0; i < 10; i++) {
       stmt.setInt(2, i);
       int rows = stmt.executeUpdate();
}
```

7.2.1 Data type conformance on IN parameters

The PreparedStatement.setXXX methods do not perform any general data type conversions. Instead the Java value is simply mapped to the corresponding SQL type (following the mapping specified in Table 3 on page 28) and that value is sent to the database.

It is the programmer’s responsibility to make sure that each argument Java type maps to a SQL type that is compatible with the SQL data type expected by the database. For maximum portability programmers should use Java types that correspond to the exact SQL types expected by the database.

If programmers require data type conversions for IN parameters, they may use the PreparedStatement.setObject method (see Section 14.2.2) which converts a Java Object to a specified SQL type before sending the value to the database.

7.2.2 Sending SQL NULLs as IN parameters

The PreparedStatement.setNull method allows you to send a SQL NULL value to the database as an IN parameter. Note however that you must specify the SQL type of the parameter.

In addition, for those setXXX methods that take Java objects as arguments, if a Java null value is passed to a setXXX method then a SQL NULL will be sent to the database.

7.2.3 Sending very large parameters

JDBC itself defines no limits on the amount of data that may be sent with a setBytes or setString 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 driver will make repeated calls on these IO streams to read their contents and transmit these as the actual parameter data.

Separate setXXX methods are provided for streams containing uninterpreted bytes, for streams containing ASCII characters, and for streams containing Unicode characters.

When setting a stream as an input parameter, the application programmer must specify the number of bytes 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(file);
java.sql.PreparedStatement stmt = conn.prepareStatement(
                    "UPDATE Table5 SET stuff = ? WHERE index = 4");
stmt.setBinaryStream(1, fin, fileLength);
// When the statement executes, the "fin" object will get called
// repeatedly to deliver up its data.
stmt.executeUpdate();
```

7.3 Receiving OUT parameters

If you are executing a stored procedure call then you should use the CallableStatement class. CallableStatement is a subtype of PreparedStatement.

To pass in any IN parameters you can use the setXXX methods defined in PreparedStatement as described in Section 7.2 above.

However if your stored procedure returns OUT parameters then for each OUT parameter you must use the CallableStatememt.registerOutParameter method to register the SQL type of the OUT parameter before you execute the statement. (See Appendix A.6.) Then after the statement has executed you must use the corresponding CallableStatement.getXXX method to retrieve the parameter value.

```
java.sql.CallableStatement stmt = conn.prepareCall(
                                "{call getTestData(?, ?)}");
stmt.registerOutParameter(java.sql.Types.TINYINT);
stmt.registerOutParameter(java.sql.types.DECIMAL, 2);
stmt.executeUpdate();
byte x = stmt.getByte(1);
Numeric n = stmt.getNumeric(2,2);
```

If a stored procedure returns one or more ResultSets in addition to returning OUT parameters, then for maximum portability you must retrieve and process the ResultSets before retrieving the OUT parameter values.

7.3.1 Data type conformance on OUT parameters

The CallableStatement.getXXX methods do not perform any general data type conversions. Instead the registerOutParameter call must specify the SQL type that will be returned by the database and the programmer must then subsequently call the getXXX method whose Java type corresponds to that SQL type, as specified in Table 2 on page 27.

7.3.2 Retrieving NULL values as OUT parameters

As with ResultSets, in order to determine if a given OUT parameter value is SQL “NULL” you must first read the parameter and then use the CallableStatement.wasNull method to discover if the read returned a SQL “NULL”.

When you read a SQL “NULL” value using one of the CallableStatement.getXXX methods, you will receive a value of null, zero, or false, following the same rules specified in section 7.1.2 for the ResultSet.getXXX methods.

7.3.3 Receiving very large out parameters

We do not provide any mechanism for retrieving OUT parameters as streams.

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

7.4 Data truncation

Under some circumstances data may be truncated when it is being read from or written to the database. How this is handled will depend on the circumstances, but in general data truncation on a database read will result in a warning whereas data truncation on a database write will result in a SQLException.

7.4.1 Exceeding the Connection maxFieldSize limit

If an application uses Connection.setMaxFieldSize to impose a limit on the maximum size of a field, then attempts to read or write a field larger than this will result in the data being silently truncated to the maxFieldSize size, without any SQLException or SQLWarning.

7.4.2 Data truncation on reads

In general data truncation errors during data reads will be uncommon with JDBC as the API does not require the programmer to pass in fixed size buffers, but rather allocates appropriate data space as needed. However in some circumstances drivers may encounter internal implementation limits, so there is still a possibility for data truncation during reads.

If data truncation occurs during a read from a ResultSet then a DataTruncation object (a subtype of SQLWarning) will get added to the ResultSet’s warning list and the method will return as much data as it was able to read. Similarly if a data truncation occurs while an OUT parameter is being received from the database then a DataTruncation object will get added to the CallableStatement’s warning list and the method will return as much data as it was able to read.

7.4.3 Data truncation on writes

During writes to the database there is a possibility that the application may attempt to send more data than the driver or the database is prepared to accept. In this case the failing method should raise a DataTruncation exception as a SQLException.

8 Mapping SQL data types into Java

8.1 Constraints

We need 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.

Table 2 shows the default Java mapping for various common SQL data types. Not all of these types will necessarily be supported by all databases. The various mappings are described more fully in the following sections.

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

Table 2: Standard mapping from SQL types to Java types.

Similarly table 3 shows the reverse mapping from Java types to SQL types.

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

Table 3: Standard mapping from Java types to SQL types.

The mapping for String will normally be VARCHAR but will turn into LONGVARCHAR if the given value exceeds the drivers limit on VARCHAR values. Similarly for byte[] and VARBINARY and LONGVARBINARY.

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.

For fixed length SQL strings of type CHAR(n) the JDBC drivers will perform appropriate padding with spaces. Thus when a CHAR(n) field is retrieved from the database the resulting String will always be of length “n” and may include some padding spaces at the end. When a

String is sent to a CHAR(n) field, the driver and/or the database will add any necessary padding spaces to the end of the String to bring it up to length “n”.

The ResultSet.getString method allocates and returns 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. Thus Java programmers can use getString 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.

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.)

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 programmers 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, 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 to Java float, and FLOAT and DOUBLE to Java double.

REAL is required to support 7 digits of mantissa precision. FLOAT and DOUBLE are required to support 15 digits of mantissa precision.

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 date 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.

Some drivers may allow more concurrent execution than others. Developers can assume fully concurrent execution; if the driver requires some form of synchronization it will provide it. The only difference visible to the developer will be that applications will run with reduced concurrency.

For example, two Statements on the same Connection can be executed concurrently and their ResultSets can be processed concurrently (from the perspective of the developer.) Some drivers will provide this full concurrency. Others may execute one statement and wait until it completes before sending the next.

One specific use of multi-threading is to cancel a long running statement. This is done by using one thread to execute the statement and another to cancel it with its Statement.cancel() method.

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.

9.3 Transactions.

New JDBC connections are initially in “auto-commit” mode. This means that each statement is executed as a separate transaction at the database.

In order to execute several statements within a single transaction you must first disable auto-commit by calling Connection.setAutoCommit(false).

When auto-commit is disabled, the connection always has an implicit transaction associated with it. You can execute a Connection.commit to complete the transaction or a Connection.rollback to abort it. The commit or rollback will also start a new implicit transaction.

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, then by default all PreparedStatements, CallableStatements, and ResultSets on the connection will be closed. Simple Statements will be left open.

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 version of the interface we will provide no support for committing transactions across different connections.

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.

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

Note that not all DBMSs support positioned update and delete. The DatabaseMetaData.supportsPositionedDelete and supportsPositionedUpdate methods can be used to discover whether a particular connection supports these operations. When they are supported, the DBMS/driver must insure that rows selected are properly locked so that positioned updates do not result in update anomalies or other concurrency problems.

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

11 SQL Extensions

Certain SQL features beyond SQL-2 Entry Level are widely supported and are desirable to include as part of our JDBC compliance definition so that applications can depend on the portability of these features. However SQL-2 Transitional Level, the next higher level of SQL compliance defined by ANSI, is not widely supported. Where Transitional Level semantics *are* supported, the syntax is often different across DBMSs.

We therefore define two kinds of extensions to SQL-2 Entry Level that must be supported by a JDBC-Compliant™ driver:

- Selective Transitional Level syntax and semantics must be supported. We currently demand just one such feature: the DROP TABLE command is required for JDBC compliance.
- Selective Transitional-Level semantics must be supported through an escape syntax that a driver can easily scan for and translate into DBMS-specific syntax. We discuss these escapes in the remainder of Section 11. Note that these escapes need only be supported where the underyling database supports the corresponding Transitional-Level semantics.

An ODBC driver that supports ODBC Core SQL as defined by Microsoft complies with JDBC SQL as defined in this section.

11.1 SQL Escape Syntax

JDBC supports the same DBMS-independent escape syntax as ODBC for stored procedures, scalar functions, dates, times, and outer joins. A driver maps this escape 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 ...}
```

This ODBC-compatible escape syntax is in general *not* the same as has been adopted by ANSI in SQL-2 Transitional Level for the same functionality. In cases where all of the desired DBMSs support the standard SQL-2 syntax, the user is encouraged to use that syntax instead of these escapes. When enough DBMSs support the more advanced SQL-2 syntax and semantics these escapes should no longer be necessary.

11.2 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. To determine if stored procedures are supported, call DatabaseMetaData.supportsStoredProcedure as described in Section 15.

11.3 Time and Date Literals

DBMSs differ in the syntax they use for date, time, and timestamp literals. 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...'}
```

The fractional seconds (.f...) portion of the TIMESTAMP can be omitted.

11.4 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.

11.5 LIKE Escape Characters

The characters "%" and "_" have special meaning in SQL LIKE clauses (to match zero or more characters, or exactly one character, respectively). In order to interpret them literally, they can be preceded with a special escape character in strings, e.g. "\". In order to specify the escape character used to quote these characters, include the following syntax on the end of the query:

```
{escape 'escape-character'}
```

For example, the query

```
SELECT NAME FROM IDENTIFIERS WHERE ID LIKE '\_%' {escape '\'}
```

finds identifier names that begin with an underbar.

11.6 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. Three boolean DatabaseMetaData methods are provided in Section 15 to determine the kinds of outer joins supported by a driver.

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, java.sql.CallableStatement, and java.sql.Resultset should all be supported for all JDBC drivers. For databases which don't support OUT parameters with stored procedures, the various registerOutParameter and getXXX methods of CallableStatement may raise SQLException.

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, in order to pass JDBC conformance tests we require at least ANSI SQL-2 Entry Level syntax and semantics plus support for the SQL extensions listed in Section 11.

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.

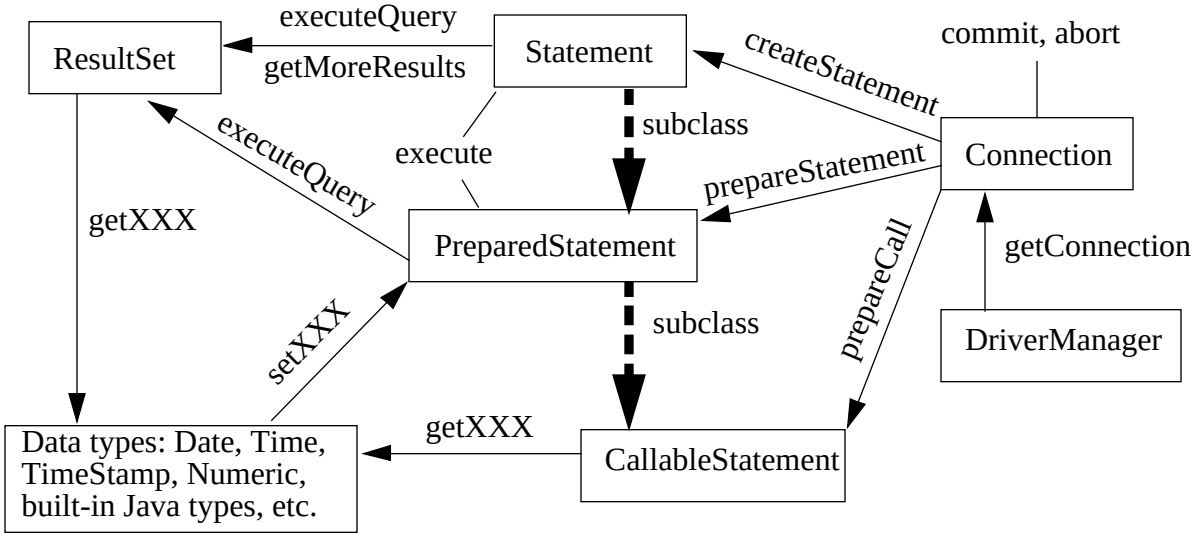
Database vendors may add additional functionality by adding new subtypes of existing JDBC types that provide additional methods. Thus the Foobah corporation might define a new Java type foobah.sql.FooBahStatement that inherits from the standard java.sql.Statement type but adds some new functionality.

13 JDBC Interface Definitions

The following pages contain the Java definitions of the core java.sql interfaces and classes. The code is organized alphabetically by class name as a convenience for the reader:

java.sql.CallableStatement
java.sql.Connection
java.sql.DataTruncation
java.sql.Date
java.sql.Driver
java.sql.DriverManager
java.sql.DriverPropertyInfo
java.sql.Numeric
java.sql.PreparedStatement
java.sql.ResultSet
java.sql.SQLException
java.sql.SQLWarning
java.sql.Statement
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) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * CallableStatement is used to execute SQL stored procedures.
 *
 * JDBC provides a stored procedure SQL escape that allows stored
 * procedures to be called in a standard way for all RDBMS's. This
 * escape syntax has one form that includes a result parameter and one
 * that does not. If used, the result parameter must be registered as
 * an OUT parameter. The other parameters may be used for input,
 * output or both. Parameters are refered to sequentially, by
 * number. The first parameter is 1.
 *
 *
 * {?= call <procedure-name>[<arg1>,<arg2>, ...]}<BR>
 * {call <procedure-name>[<arg1>,<arg2>, ...]}
 *
 *
 * IN parameter values are set using the set methods inherited from
 * PreparedStatement. The type of all OUT parameters must be
 * registered prior to executing the stored procedure; their values
 * are retrieved after execution via the get methods provided here.
 *
 * A Callable statement may return a ResultSet or multiple
 * ResultSets. Multiple ResultSets are handled using operations
 * inherited from Statement.
 *
 */
public interface CallableStatement extends PreparedStatement {

    /**
     * Before executing a stored procedure call you must explicitly
     * call registerOutParameter to register the java.sql.Type of each
     * out parameter.
     *
     * Note: When reading the value of an out parameter you
     * must use the getXXX method whose Java type XXX corresponds to the
     * parameter's registered SQL type.
     *
     * @param parameterIndex the first parameter is 1, the second is 2,...
     * @param sqlType SQL type code defined by java.sql.Types;
     * for parameters of type Numeric or Decimal use the version of
     * registerOutParameter that accepts a scale value
     */
    void registerOutParameter(int parameterIndex, int sqlType)
        throws SQLException;

    /**
     * Use this version of registerOutParameter for registering
     * Numeric or Decimal out parameters.
     *
     * Note: When reading the value of an out parameter you
```

```
     * must use the getXXX method whose Java type XXX corresponds to the
     * parameter's registered SQL type.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param sqlType use either java.sql.Type.NUMERIC or java.sql.Type.DECIMAL
     * @param scale a value greater than or equal to zero representing the
     *              desired number of digits to the right of the decimal point
     */
    void registerOutParameter(int parameterIndex, int sqlType, int scale)
        throws SQLException;

    /**
     * An OUT parameter may have the value of SQL NULL; wasNull reports
     * whether the last value read has this special value.
     *
     * Note: You must first call getXXX on a parameter to
     * read its value and then call wasNull() to find if the value was
     * SQL NULL.
     *
     * @return true if the last parameter read was SQL NULL
     */
    boolean wasNull() throws SQLException;

    /**
     * Get the value of a CHAR, VARCHAR, or LONGVARCHAR parameter as a Java String.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @return the parameter value; if the value is SQL NULL the result is null
     */
    String getString(int parameterIndex) throws SQLException;

    /**
     * Get the value of a BIT parameter as a Java boolean.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @return the parameter value; if the value is SQL NULL the result is false
     */
    boolean getBoolean(int parameterIndex) throws SQLException;

    /**
     * Get the value of a TINYINT parameter as a Java byte.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @return the parameter value; if the value is SQL NULL the result is 0
     */
    byte getByte(int parameterIndex) throws SQLException;

    /**
     * Get the value of a SMALLINT parameter as a Java short.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @return the parameter value; if the value is SQL NULL the result is 0
     */
    short getShort(int parameterIndex) throws SQLException;
```

```
/**
 * Get the value of a INTEGER parameter as a Java int.
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is 0
 */
int getInt(int parameterIndex) throws SQLException;

/**
 * Get the value of a BIGINT parameter as a Java long.
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is 0
 */
long getLong(int parameterIndex) throws SQLException;

/**
 * Get the value of a FLOAT parameter as a Java float.
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is 0
 */
float getFloat(int parameterIndex) throws SQLException;

/**
 * Get the value of a DOUBLE parameter as a Java double.
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is 0
 */
double getDouble(int parameterIndex) throws SQLException;

/**
 * Get the value of a NUMERIC parameter as a java.sql.Numeric object.
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is null
 */
Numeric getNumeric(int parameterIndex, int scale) throws SQLException;

/**
 * Get the value of a SQL BINARY or VARBINARY parameter as a Java byte[]
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is null
 */
byte[] getBytes(int parameterIndex) throws SQLException;

/**
 * Get the value of a SQL DATE parameter as a java.sql.Date object
 *
 * @param parameterIndex the first parameter is 1, the second is 2, ...
 * @return the parameter value; if the value is SQL NULL the result is null
```

```
        */
       java.sql.Date getDate(int parameterIndex) throws SQLException;

       /**
        * Get the value of a SQL TIME parameter as a java.sql.Time object.
        *
        * @param parameterIndex the first parameter is 1, the second is 2, ...
        * @return the parameter value; if the value is SQL NULL the result is null
        */
       java.sql.Time getTime(int parameterIndex) throws SQLException;

       /**
        * Get the value of a SQL TIMESTAMP parameter as a java.sql.Timestamp object.
        *
        * @param parameterIndex the first parameter is 1, the second is 2, ...
        * @return the parameter value; if the value is SQL NULL the result is null
        */
       java.sql.Timestamp getTimestamp(int parameterIndex)
           throws SQLException;

       // Advanced features:

       /**
        * Get the value of a parameter as a Java object.
        *
        * This method returns a Java object whose type coresponds to the SQL
        * type that was registered for this parameter using registerOutParameter.
        *
        * Note that this method may be used to read datatabase specific abstract
        * data types, by specifying a targetSqlType of java.sql.types.OTHER which
        * allows the driver to return a database specific Java type.
        *
        * @param parameterIndex The first parameter is 1, the second is 2, ...
        * @return A java.lang.Object holding the OUT parameter value.
        */
       Object getObject(int parameterIndex) throws SQLException;

   }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * A Connection represents a session with a specific
 * database. Within the context of a Connection, SQL statements are
 * executed and results are returned.
 *
 * A Connection's database is able to provide information
 * describing its tables, its supported SQL grammar, its stored
 * procedures, the capabilities of this connection, etc. This
 * information is obtained with the getMetaData method.
 *
 * Note: By default the Connection automatically commits
 * changes after executing each statement. If auto commit has been
 * disabled an explicit commit must be done or database changes will
 * not be saved.
 *
 */
public interface Connection {

    /**
     * SQL statements without parameters are normally
     * executed using Statement objects. If the same SQL statement
     * is executed many times it is more efficient to use a
     * PreparedStatement
     *
     * @return a new Statement object
     */
    Statement createStatement() throws SQLException;

    /**
     * A SQL statement with or without IN parameters can be
     * pre-compiled and stored in a PreparedStatement object. This
     * object can then be used to efficiently execute this statement
     * multiple times.
     *
     * Note: This method is optimized for handling
     * parametric SQL statements that benefit from precompilation. If
     * the driver supports precompilation, prepareStatement will send
     * the statement to the database for precompilation. Some drivers
     * may not support precompilation. In this case, the statement may
     * not be sent to the database until the PreparedStatement is
     * executed.  This has no direct affect on users; however, it does
     * affect which method throws certain SQLExceptions.
     *
     * @param sql a SQL statement that may contain one or more '?' IN
     * parameter placeholders
     *
     * @return a new PreparedStatement object containing the
     * pre-compiled statement
     */
    PreparedStatement prepareStatement(String sql)
```

```
    throws SQLException;

/**
 * A SQL stored procedure call statement is handled by creating a
 * CallableStatement for it. The CallableStatement provides
 * methods for setting up its IN and OUT parameters, and
 * methods for executing it.
 *
 * Note: This method is optimized for handling stored
 * procedure call statements. Some drivers may send the call
 * statement to the database when the prepareCall is done; others
 * may wait until the CallableStatement is executed. This has no
 * direct affect on users; however, it does affect which method
 * throws certain SQLExceptions.
 *
 * @param sql a SQL statement that may contain one or more '?'
 * parameter placeholders
 *
 * @return a new CallableStatement object containing the
 * pre-compiled SQL statement
 */
CallableStatement prepareCall(String sql) throws SQLException;

/**
 * A driver may convert the JDBC sql grammar into its system's
 * native SQL grammar prior to sending it; nativeSQL returns the
 * native form of the statement that the driver would have sent.
 *
 * @param sql a SQL statement that may contain one or more '?'
 * parameter placeholders
 *
 * @return the native form of this statement
 */
String nativeSQL(String sql) throws SQLException;

/**
 * If a connection is in auto-commit mode, then all its SQL
 * statements will be executed and committed as individual
 * transactions.  Otherwise, its SQL statements are grouped into
 * transactions that are terminated by either commit() or
 * rollback().  By default, new connections are in auto-commit
 * mode.
 *
 * @param autoCommit true enables auto-commit; false disables
 * auto-commit.
 */
void setAutoCommit(boolean autoCommit) throws SQLException;

/**
 * Get the current auto-commit state.
 * @return Current state of auto-commit mode.
 */
boolean getAutoCommit() throws SQLException;
```

```
/**
 * Commit makes all changes made since the previous
 * commit/rollback permanent and releases any database locks
 * currently held by the Connection.
 *
 * Note: By default, a Connection's PreparedStatements,
 * CallableStatements and ResultSets are implicitly closed when it
 * is committed.
 *
 */
void commit() throws SQLException;

/**
 * Rollback drops all changes made since the previous
 * commit/rollback and releases any database locks currently held
 * by the Connection.
 *
 * Note: By default, a Connection's PreparedStatements,
 * CallableStatements and ResultSets are implicitly closed when it
 * is committed.
 *
 */
void rollback() throws SQLException;

/**
 * In some cases, it is desirable to immediately release a
 * Connection's database and JDBC resources instead of waiting for
 * them to be automatically released; the close method provides this
 * immediate release.
 *
 * Note: A Connection is automatically closed when it is
 * garbage collected. Certain fatal errors also result in a closed
 * Connection.
 */
void close() throws SQLException;;

/**
 * Check if a Connection is closed
 *
 * @return true if the connection is closed; false if it's still open
 */
boolean isClosed() throws SQLException;;

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

/**
 * A Connection's database is able to provide information
 * describing its tables, its supported SQL grammar, its stored
 * procedures, the capabilities of this connection, etc. This
 * information is made available through a DatabaseMetaData
 * object.
 *
 * @return a DatabaseMetaData object for this Connection
```

```
     */
    DatabaseMetaData getMetaData() throws SQLException;;

    /**
     * You can put a connection in read-only mode as a hint to enable
     * database optimizations.
     *
     * Note: setReadOnly cannot be called while in the
     * middle of a transaction.
     *
     * @param readOnly true enables read-only mode; false disables
     * read-only mode.
     */
    void setReadOnly(boolean readOnly) throws SQLException;

    /**
     * Test if the connection is in read-only mode
     *
     * @return true if connection is read-only
     */
    boolean isReadOnly() throws SQLException;

    /**
     * A sub-space of this Connection's database may be selected by setting a
     * catalog name. If the driver does not support catalogs it will
     * silently ignore this request.
     */
    void setCatalog(String catalog) throws SQLException;

    /**
     * Return the Connection's current catalog name
     *
     * @return the current catalog name or null
     */
    String getCatalog() throws SQLException;

    /**
     * Transactions are not supported.
     */
    int TRANSACTION_NONE     = 0;

    /**
     * Dirty reads are done
     */
    int TRANSACTION_READ_UNCOMMITTED = 1;

    /**
     * Only reads on the current row are repeatable
     */
    int TRANSACTION_READ_COMMITTED   = 2;

    /**
     * Reads on all rows of a result are repeatable
     */
```

```
int TRANSACTION_REPEATABLE_READ  = 4;

/**
 * Reads on all rows of a transaction are repeatable
 */
int TRANSACTION_SERIALIZABLE     = 8;

/**
 * You can call this method to try to change the transaction
 * isolation level on a newly opened connection, using one of the
 * TRANSACTION_* values.
 *
 * Note: setTransactionIsolation cannot be called while
 * in the middle of a transaction.
 *
 * @param level one of the TRANSACTION_* isolation values with the
 * exception of TRANSACTION_NONE; some databases may not support
 * other values
 *
 */
void setTransactionIsolation(int level) throws SQLException;

/**
 * Get this Connection's current transaction isolation mode
 *
 * @return the current TRANSACTION_* mode value
 */
int getTransactionIsolation() throws SQLException;

/**
 * When a Connection is in auto-close mode all its
 * PreparedStatements, CallableStatements, and ResultSets will be
 * closed when a transaction is committed or rolled back.  By
 * default, a new Connection is in auto-close mode.
 *
 *  When auto-close is disabled JDBC attempts to keep
 * all statements and ResultSets open across commits and
 * rollbacks.  However the actual behaviour will vary depending
 * on what the underlying database supports.  Some databases
 * allow these objects to remain open across commits whereas
 * other databases insist on closing them.
 *
 * @param autoClose true enables auto-close, false disables
 *  auto-close.
 *
 */
void setAutoClose(boolean autoClose) throws SQLException;

/**
 * Get the current auto-close state.
 * @return Current state of auto-close mode.
 */
boolean getAutoClose() throws SQLException;
```

```
    /**
     * The first warning reported by calls on this Connection is
     * returned.
     *
     * Note: Subsequent warnings will be chained to this
     * SQLWarning.
     *
     * @return the first SQLWarning or null
     */
    SQLWarning getWarnings() throws SQLException;

    /**
     * After this call getWarnings returns null until a new warning is
     * reported for this Connection.
     */
    void clearWarnings() throws SQLException;
}
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * When JDBC unexpectedly truncates a data value it reports a
 * DataTruncation warning (on reads) or throws a DataTruncation exception
 * (on writes).
 *
 * The SQLstate for a DataTruncation is set to "01004".
 */

public class DataTruncation extends SQLWarning {

    public DataTruncation(int index, boolean parameter,
          boolean read, int dataSize,
          int transferSize);

    /**
     * Get the index of the column or parameter that was truncated.
     *
     * This may be -1 if the column or parameter index is unknown, in
     * which case the "parameter" and "read" fields should be ignored.
     *
     * @return the DataTruncation's index value
     */
    public int getIndex();

    /**
     * True if the value was a parameter; false if it was a column value.
     *
     * @return the DataTruncation's parameter value
     */
    public boolean getParameter();

    /**
     * True if the value was truncated when read from the database; false
     * if the data was truncated on a write.
     *
     * @return the DataTruncation's read value
     */
    public boolean getRead();

    /**
     * Get the number of bytes of data that should have been transferred.
     * This number may be approximate if data conversions were being
     * performed.  The value may be "-1" if the size is unknown.
     *
     * @return the DataTruncation's dataSize value
     */
    public int getDataSize();

    /**
     * Get the number of bytes of data actually transferred.
```

```
        * The value may be “-1” if the size is unknown.
        *
        * @return the DataTruncation’s transferSize value
        */
       public int getTransferSize();

   }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

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

    /**
     * Construct a Date
     *
     * @param year year-1900
     * @param month 0 to 11
     * @param day 1 to 31
     */
    public Date(int year, int month, int day);

    /**
     * Convert a formatted string to a Date value
     *
     * @param s date in format “yyyy-mm-dd”
     * @return corresponding Date
     */
    public static Date valueOf(String s);

    /**
     * Format a date as yyyy-mm-dd
     *
     * @return a formatted date String
     */
    public String toString ();

}
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * The Java SQL framework allows for multiple database drivers.
 *
 * Each driver should supply a driver class that implements
 * the Driver interface.
 *
 * The DriverManager 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 instantiated it should register itself
 * with the SQL framework by calling DriverManager.registerDriver
 *
 * Note: Each driver must support a null constructor so it can be
 * instantiated by doing:
 *
 *     java.sql.Driver d = Class.forName("foo.bah.Driver").newInstance();
 *
 */
public interface Driver {

    /**
     * Try to make a database connection to the given URL.
     * The driver should return "null" if it realizes it is the wrong kind
     * of 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.
     *
     * @param url The URL of the database to connect to
     *
     * @param info a list of arbitrary string tag/value pairs as
     * connection arguments; normally at least a "user" and
     * "password" property should be included
     *
     * @return a Connection to the URL
     */
    Connection connect(String url, java.util.Properties info)
```

```
    throws SQLException;

/**
 * Returns true if the driver thinks that it can open a connection
 * to the given URL.  Typically drivers will return true if they
 * understand the subprotocol specified in the URL and false if
 * they don't.
 *
 * @param url The URL of the database.
 * @return True if this driver can connect to the given URL.
 */
boolean acceptsURL(String url) throws SQLException;

/**
 * The getPropertyInfo method is intended to allow a generic GUI tool to
 * discover what properties it should prompt a human for in order to get
 * enough information to connect to a database.  Note that depending on
 * the values the human has supplied so far, additional values may become
 * necessary, so it may be necessary to iterate though several calls
 * to getPropertyInfo.
 *
 * @param url The URL of the database to connect to.
 * @param info A proposed list of tag/value pairs that will be sent on
 *          connect open.
 * @return An array of DriverPropertyInfo objects describing possible
 *          properties.  This array may be an empty array if no properties
 *          are required.
 */
DriverPropertyInfo[] getPropertyInfo(String url, java.util.Properties info)
     throws SQLException;

/**
 * Get the driver's major version number. Initially this should be 1.
 */
int getMajorVersion();

/**
 * Get the driver's minor version number. Initially this should be 0.
 */
int getMinorVersion();

/**
 * Report whether the Driver is a genuine JDBC COMPLIANT (tm) driver.
 * A driver may only report "true" here if it passes the JDBC compliance
 * tests, otherwise it is required to return false.
 *
 * JDBC compliance requires full support for the JDBC API and full support
 * for SQL 92 Entry Level.  It is expected that JDBC compliant drivers will
 * be available for all the major commercial databases.
 *
 * This method is not intended to encourage the development of non-JDBC
 * compliant drivers, but is a recognition of the fact that some vendors
 * are interested in using the JDBC API and framework for lightweight
 * databases that do not support full database functionality, or for
```

```
         * special databases such as document information retrieval where a SQL
         * implementation may not be feasible.
         */
        boolean jdbcCompliant();
    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * The DriverManager class provides access to global SQL state.
 *
 * As part of its initialization the DriverManager class will use the
 * system "jdbc.drivers" property.  This should contain a list of
 * classnames for driver classes.  The DriverManager class will attempt
 * to load each of these driver classes.  For example in your
 * ~/.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
 * DriverManager 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.
 *
 */
public class DriverManager {

    /**
     * Attempt to establish a connection to the given database URL.
     * The DriverManager attempts to select an appropriate driver from
     * the set of registered JDBC drivers.
     *
     * @param url a database url of the form jdbc:subprotocol:subname
     * @param info a list of arbitrary string tag/value pairs as
     * connection arguments; normally at least a "user" and
     * "password" property should be included
     * @return a Connection to the URL
     */
    public static synchronized Connection getConnection(String url,
            java.util.Properties info) throws SQLException;

    /**
     * Attempt to establish a connection to the given database URL.
     * The DriverManager attempts to select an appropriate driver from
     * the set of registered JDBC drivers.
     *
     * @param url a database url of the form jdbc:subprotocol:subname
     * @param user the database user on whom's behalf the Connection is being made
     * @param password the user's password
     * @return a Connection to the URL
     */
    public static synchronized Connection getConnection(String url,
                String user, String password) throws SQLException;

    /**
     * Attempt to establish a connection to the given database URL.
     * The DriverManager attempts to select an appropriate driver from
     * the set of registered JDBC drivers.
```

```
     *
     * @param url a database url of the form jdbc:subprotocol:subname
     * @return a Connection to the URL
     */
    public static synchronized Connection getConnection(String url)
                                    throws SQLException;

    /**
     * Attempt to locate a driver that understands the given URL.
     * The DriverManager attempts to select an appropriate driver from
     * the set of registered JDBC drivers.
     *
     * @param url a database url of the form jdbc:subprotocol:subname
     * @return a Driver that can connect to the URL
     */
    public static Driver getDriver(String url) throws SQLException;

    /**
     * A newly loaded driver class should call registerDriver to make itself
     * known to the DriverManager.
     *
     * @param driver the new JDBC Driver
     */
    public static synchronized void registerDriver(java.sql.Driver driver)
                        throws SQLException;

    /**
     * Drop a Driver from the DriverManager's list.  Applets can only
     * deregister Drivers from their own classloader.
     *
     * @param driver the JDBC Driver to drop
     */
    public static void deregisterDriver(Driver driver) throws SQLException;

    /**
     * Return an Enumeration of all the currently loaded JDBC drivers
     * which the current caller has access to.
     *
     * Note: The classname of a driver can be found using
     *     d.getClass().getName()
     *
     * @return the list of JDBC Driver's loaded by the caller's class loader
     */
    public static java.util.Enumeration getDrivers();

    /**
     * Set the maximum time in seconds that all drivers can wait
     * when attempting to login to a database.
     *
     * @param seconds the driver login time limit
     */
    public static void setLoginTimeout(int seconds);

    /**
```

```
         * Get the maximum time in seconds that all drivers can wait
         * when attempting to login to a database.
         *
         * @return the driver login time limit
         */
        public static int getLoginTimeout();

        /**
         * Set the logging/tracing PrintStream that is used by the DriverManager
         * and all drivers.
         *
         * @param out the new logging/tracing PrintStream; to disable, set to null
         */
        public static void setLogStream(java.io.PrintStream out);

        /**
         * Get the logging/tracing PrintStream that is used by the DriverManager
         * and all drivers.
         *
         * @return the logging/tracing PrintStream; if disabled, is null
         */
        public static java.io.PrintStream getLogStream();

        /**
         * Print a message to the current JDBC log stream
         *
         * @param message a log or tracing message
         */
        public synchronized static void println(String message);

        // Class initialization.
        static void initialize();

    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * The DriverPropertyInfo class is only of interest to advanced programmers
 * who need to interact with a Driver via getDriverProperties to discover
 * and supply properties for connections.
 */

public class DriverPropertyInfo {

    /**
     * Initial constructor.  The name is the name of the property and
     * the value is the current value, which may be null.
     */
    public DriverPropertyInfo(String name, String value);

    /**
     * The name of the property.
     */
    public String name;

    /**
     * A brief description of the property.  This may be null.
     */
    public String description = null;

    /**
     * “required” is true if a value must be supplied for this property
     * during Driver.connect.  Otherwise the property is optional.
     */
    public boolean required = false;

    /**
     * “value” specifies the current value of the property, based on a
     * combination of the information supplied to getPropertyInfo, the
     * Java environment, and driver supplied default values.  This
     * may be null if no value is known.
     */
    public String value = null;

    /**
     * If the value may be selected from a particular set of values,
     * then this is an array of the possible values.  Otherwise it should
     * be null.
     */
    public String[] choices = null;
}
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * The java.sql.Numeric class is an arbitrary precision and scale
 * number class that can be used to represent SQL fixed point NUMERIC
 * and DECIMAL values.
 *
 * Numeric objects are composed of four properties:
 * Precision : The total number of digits comprising the number,
 * including all the digits to the right and left of the decimal
 * point.
 * Scale: The number of digits to the right of the decimal point.
 * Sign:  Positive or negative.
 * Value: A vector of digits of size <I>precision</I>.
 *
 *
 * The theoretical limit to the precision and scale values is 0x7fffffff.
 * Numeric objects are immutable.
 */

public final class Numeric extends java.lang.Number {

    /**
     * Sets the value of the roundingValue which is used in rounding
     * operations.  The roundingValue is a digit between 0 and 9 that
     * determines rounding behavior.  When the scale of a Numeric is
     * reduced either by request or in a calculation, rounding is
     * performed if the value of the scale + 1 digit to the right of
     * the decimal point is greater than the rounding index.
     *
     * For example, if the number is 1.995, and the scale is set to
     * 2 and the roundingValue is 4, then the number would be rounded
     * to 2.00.  The default roundingValue is 4. A
     * roundingValue of 9 cancels rounding.
     *
     * @param val The value between 0 and 9 of the desired roundingValue.
     */
    public static void setRoundingValue(int val);

    /**
     * Returns the value of the roundingValue which is used in
     * rounding operations.  The roundingValue is a digit between 0
     * and 9 that determines rounding behavior.  When the scale of a
     * Numeric is reduced either by request or in a calculation,
     * rounding is performed if the value of the scale + 1 digit to
     * the right of the decimal point is greater than the rounding
     * index.
     *
     * For example, if the number is 1.995, and the scale is set to
     * 2 and the RoundingValue is 4, then the number would be rounded
     * to 2.00.  The default roundingValue is 4. A
     * roundingValue of 9 cancels rounding.
```

```
     *
     * @return rounding value
     */
    public static int getRoundingValue();

    /**
     * Creates a numeric from a string. The string may contain a sign
     * and a decimal point, e.g "-55.12".  Spaces and other
     * non-numeric characters are ignored.
     *
     * @param s the string used to initialize the Numeric.
     */

    public Numeric(String s);

    /**
     * Construct a Numeric given a String and an integer scale.  The
     * scale represents the desired number of places to the right of
     * the decimal point.
     *
     * The String may contain a sign and a decimal point,
     * e.g. "-55.12".  The scale must be an integer with value greater
     * than or equal to zero.  If the scale differs from the implicit
     * decimal point given in the String the value is adjusted to the
     * given scale.  This may involve rounding.
     *
     * For example, Numeric("99.995",2) would result in a numeric
     * with a scale of 2 and a value of "100.00".  If the String
     * contains no decimal point, then the scale is determined solely
     * by the given integer scale. No implicit decimal point is
     * assumed.  For example, the constructor Numeric("123",2) creates
     * a Numeric with a scale of 2, a precision of 5, and a value of
     * "123.00".
     *
     * Note: Rounding is controlled by the roundingIndex
     * function.
     *
     * @param  s    the string used to initialize the Numeric.
     * @param scale the value greater than or equal to zero
     * representing the desired number of digits to the right of the
     * decimal point.
     */
    public Numeric(String s,int scale);

    /**
     * Construct a Numeric from an integer given an integer value and
     * an integer scale.  The scale is set to the value specified. No
     * implicit decimal point is assumed, i.e., if the double value is
     * "1234" and the scale 2, the resulting numeric will be "1234.00.
     *
     * @param  x    The double value used to initialize the new Numeric.
     * @param scale The desired number of digits after the decimal point.
     */
    public Numeric(int x,int scale);
```

```
/**
 * Construct a Numeric from a double given a double value and an
 * integer scale.  The scale is set to the value specified. No
 * implicit decimal point is assumed, i.e., if the double value is
 * “1234” and the scale 2, the resulting numeric will be “1234.00.
 * If the double value is “1234.5” and the scale 2 the value will
 * be “1234.50”.  Rounding may occur during this conversion.
 *
 * @param  x    The double value used to initialize the new Numeric.
 * @param scale The desired number of digits after the decimal point.
 */
public Numeric(double x, int scale);

/**
 * Construct a Numeric from another Numeric.  The precision,
 * scale, and value of the new Numeric are copied from the Numeric
 * given in the argument.
 *
 * @param x The Numeric used to initialize the new Numeric.
 */
public Numeric(Numeric x);

/**
 * Given an existing Numeric and an integer scale value, construct
 * a new Numeric with the scale adjusted accordingly.  The
 * precision, scale, and value of the new Numeric are copied from
 * the argument Numeric. The scale is then adjusted to the scale
 * given as a parameter. This may result in rounding of the value,
 * as well as a change to the precision.
 *
 * @param  xThe Numeric to copy.
 * @param scale An integer representing the desired * scale of the
 * new Numeric.
 */
public Numeric(Numeric x, int scale);

/**
 * The following methods convert the Numeric value to an
 * appropriate Java built-in type.  Note that these conversions
 * may result in a loss of precision.
 *
 */

/**
 * Convert the Numeric value to the Java built-in type of int.
 * This conversion may result in a loss of precision.  Digits
 * after the decimal point are dropped.
 *
 * @return An integer representation of the Numeric value.
 */
public int intValue();

/**
```

```
     * Convert the Numeric value to the Java built-in type of long.
     * This conversion may result in a loss of precision.  Digits
     * after the decimal point are dropped.
     *
     * @return A long integer representation of the Numeric value.
     */
    public long longValue();

    /**
     * Convert the Numeric value to the Java built-in type of float.
     * This conversion may result in a loss of precision.
     *
     * @return A float representation of the Numeric value.
     */
    public float floatValue();

    /**
     * Convert the Numeric value to the Java built-in type of double.
     * This conversion may result in a loss of precision.
     *
     * @return A double representation of the Numeric value.
     */
    public  double doubleValue();

    /**
     * Convert the Numeric value to the Java built-in type of String.
     * Negative numbers are represented by a leading "-". No sign is
     * added for positive numbers.
     *
     * @return A String representation of the Numeric value.
     */
    public  String toString();

    /**
     * Return the number of digits to the right of the decimal point.
     *
     * @return An integer value representing the number of decimal
     * places to the right of the decimal point.
     */
    public int getScale();

    /**
     * Return the value multiplied by 10**scale. Precision may be
     * lost.  Thus, if a currency value was being kept with scale "2"
     * as "5.04", the getScaled function would return the long integer
     * "504".
     *
     * @return The scaled value as a long.
     */
    public long getScaled();

    /**
     * Return a numeric value created by dividing the argument by
     * 10**scale.  Thus, createFromScaled(504,2) would create the
```

```
     * value “5.04”.
     *
     * @param  scaled The scaled value as a long.
     * @param  s      The desired scale value as an integer between 0 and 9
     * @return A new numeric.
     */
    public static Numeric createFromScaled(long scaled, int s);
    /**
     * Returns the arithmetic sum of the Numeric and the argument.
     * The scale of the sum is determined by this object not by the
     * argument.
     *
     * @param  n  The Numeric to add.
     * @return The sum as a Numeric.
     */
    public Numeric add(Numeric n);

    /**
     * Returns the arithmetic difference between the Numeric and the
     * argument.  The scale of the sum is determined by this object
     * not by the argument.
     *
     * @param  n The Numeric to subtract.
     * @return The difference as a Numeric.
     */
    public Numeric subtract(Numeric n);

    /**
     * Returns the arithmetic product of the object Numeric and the
     * argument.  The scale of the product is determined by this object
     * not by the argument.
     *
     * @param  x The multiplier as a Numeric.
     * @return The product as a Numeric.
     */
    public Numeric multiply(Numeric x);

    /**
     * Returns the arithmentic quotient calculated by dividing the
     * Numeric by the argument.  The scale of the quotient is
     * determined by this object not by the argument.
     *
     * @param  x The divisor as a Numeric.
     * @return The quotient as a Numeric.
     */
    public Numeric divide(Numeric x);

    /**
     * Returns true if the arithmetic value of the Numeric equals the
     * argument.
     *
     * @param  obj The object to compare.
     * @return The boolean result of the comparison.
     */
```

```
public  boolean equals(Object obj);

/**
 * Returns true if the arithmetic value of the Numeric is less
 * than the argument.
 *
 * @param  x The object to compare.
 * @return The boolean result of the comparison.
 */
public boolean lessThan(Numeric x);

/**
 * Returns true if the arithmetic value of the Numeric is less
 * than or equals the argument.
 *
 * @param  x The object to compare.
 * @return The boolean result of the comparison.
 */
public  boolean lessThanOrEquals(Numeric x);

/**
 * Returns true if the arithmetic value of the Numeric is greater
 * than the argument.
 *
 * @param  x The object to compare.
 * @return The boolean result of the comparison.
 */
public  boolean greaterThan(Numeric x);

/**
 * Returns true if the arithmetic value of the Numeric is greater
 * than or equals the argument.
 *
 * @param  x The object to compare.
 * @return The boolean result of the comparison.
 */
public  boolean greaterThanOrEquals(Numeric x);

/**
 * Returns an integer hashcode for the Numeric.  Note that the
 * code returned for 03.00 will equal the code for 3.0.
 *
 * @return The hashcode as an integer.
 */
public int hashCode();

/**
 * Returns a numeric copied from the current object with the scale
 * adjusted as specified by the argument.  Sets the number of the
 * digits to the right of the decimal point.  The precision and
 * value of the Numeric will be adjusted accordingly.  Note that
 * changing the scale to a smaller value may result in rounding
 * the value.
 *
```

```
         * @param  scale the character to be converted
         * @return A Numeric with the adjusted scale.
         */
        public Numeric setScale(int scale);

    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * A SQL statement is pre-compiled and stored in a
 * PreparedStatement object. This object can then be used to
 * efficiently execute this statement multiple times.
 *
 * Note: The setXXX methods for setting IN parameter values
 * must specify types that are compatible with the defined SQL type of
 * the input parameter. For instance, if the IN parameter has SQL type
 * Integer then setInt should be used.
 *
 * If arbitrary parameter type conversions are required then the
 * setObject method should be used with a target SQL type.
 *
 */

public interface PreparedStatement extends Statement {

    /**
     * A prepared SQL query is executed and its ResultSet is returned.
     *
     * @return a ResultSet that contains the data produced by the query
     */
    ResultSet executeQuery() throws SQLException;

    /**
     * Execute a SQL INSERT, UPDATE or DELETE statement. In addition,
     * SQL statements that return nothing such as SQL DDL statements
     * can be executed.
     *
     * @return either the row count for INSERT, UPDATE or DELETE; or 0
     * for SQL statements that return nothing
     */
    int executeUpdate() throws SQLException;

    /**
     * Set a parameter to SQL NULL.
     *
     * Note: You must specify the parameter's SQL type.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param sqlType SQL type code defined by java.sql.Types
     */
    void setNull(int parameterIndex, int sqlType) throws SQLException;

    /**
     * Set a parameter to a Java boolean value.  The driver converts this
     * to a SQL BIT value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
```

```
     */
    void setBoolean(int parameterIndex, boolean x) throws SQLException;

    /**
     * Set a parameter to a Java byte value.  The driver converts this
     * to a SQL TINYINT value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setByte(int parameterIndex, byte x) throws SQLException;

    /**
     * Set a parameter to a Java short value.  The driver converts this
     * to a SQL SMALLINT value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setShort(int parameterIndex, short x) throws SQLException;

    /**
     * Set a parameter to a Java int value.  The driver converts this
     * to a SQL INTEGER value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setInt(int parameterIndex, int x) throws SQLException;

    /**
     * Set a parameter to a Java long value.  The driver converts this
     * to a SQL BIGINT value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setLong(int parameterIndex, long x) throws SQLException;

    /**
     * Set a parameter to a Java float value.  The driver converts this
     * to a SQL FLOAT value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setFloat(int parameterIndex, float x) throws SQLException;

    /**
     * Set a parameter to a Java double value.  The driver converts this
     * to a SQL DOUBLE value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
```

```
     */
    void setDouble(int parameterIndex, double x) throws SQLException;

    /**
     * Set a parameter to a java.sql.Numeric value.  The driver converts this
     * to a SQL NUMERIC value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setNumeric(int parameterIndex, Numeric x) throws SQLException;

    /**
     * Set a parameter to a Java String value.  The driver converts this
     * to a SQL VARCHAR or LONGVARCHAR value (depending on the arguments
     * size relative to the driver's limits on VARCHARs) when it sends
     * it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setString(int parameterIndex, String x) throws SQLException;

    /**
     * Set a parameter to a Java array of bytes.  The driver converts this
     * to a SQL VARBINARY or LONGVARBINARY (depending on the arguments
     * size relative to the driver's limits on VARBINARYs) when it sends
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setBytes(int parameterIndex, byte x[]) throws SQLException;

    /**
     * Set a parameter to a java.sql.Date value.  The driver converts this
     * to a SQL DATE value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setDate(int parameterIndex, java.sql.Date x)
        throws SQLException;

    /**
     * Set a parameter to a java.sql.Time value.  The driver converts this
     * to a SQL TIME value when it sends it to the database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setTime(int parameterIndex, java.sql.Time x)
        throws SQLException;

    /**
```

```
     * Set a parameter to a java.sql.Timestamp value.  The driver
     * converts this to a SQL TIMESTAMP value when it sends it to the
     * database.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the parameter value
     */
    void setTimestamp(int parameterIndex, java.sql.Timestamp x)
        throws SQLException;

    /**
     * When a very large ASCII value is input to a LONGVARCHAR
     * parameter it may be more practical to send it via a
     * java.io.InputStream. JDBC will read the data from the stream
     * as needed, until it reaches end-of-file.  The JDBC driver will
     * do any necessary conversion from ASCII to the database char format.
     *
     * Note: this stream object can either be a standard
     * Java stream object, or your own subclass that implements the
     * standard interface.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the java input stream which contains the ASCII parameter value
     * @param length the number of bytes in the stream
     */
    void setAsciiStream(int parameterIndex, java.io.InputStream x, int length)
        throws SQLException;

    /**
     * When a very large UNICODE value is input to a LONGVARCHAR
     * parameter it may be more practical to send it via a
     * java.io.InputStream. JDBC will read the data from the stream
     * as needed, until it reaches end-of-file.  The JDBC driver will
     * do any necessary conversion from UNICODE to the database char format.
     *
     * Note: this stream object can either be a standard
     * Java stream object, or your own subclass that implements the
     * standard interface.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the java input stream which contains the
     * UNICODE parameter value @param length the number of bytes in
     * the stream
     */
    void setUnicodeStream(int parameterIndex, java.io.InputStream x, int length)
        throws SQLException;

    /**
     * When a very large binary value is input to a LONGVARBINARY
     * parameter it may be more practical to send it via a
     * java.io.InputStream. JDBC will read the data from the stream
     * as needed, until it reaches end-of-file.
     *
     * Note: this stream object can either be a standard
```

```
     * Java stream object, or your own subclass that implements the
     * standard interface.
     *
     * @param parameterIndex the first parameter is 1, the second is 2, ...
     * @param x the java input stream which contains the binary parameter value
     * @param length the number of bytes in the stream
     */
    void setBinaryStream(int parameterIndex, java.io.InputStream x, int length)
        throws SQLException;

    /**
     * In general parameter values remain in force for repeated use of a
     * Statement. Setting a parameter value automatically clears its
     * previous value.  However In some cases it is useful to immediately
     * release the resources used by the current parameter values; this can
     * be done by calling clearParameters.
     */
    void clearParameters() throws SQLException;

    // Advanced features:

    /**
     * Set the value of a parameter using an object; use the
     * java.lang equivalent objects for integral values.
     *
     * The given Java object will be converted to the targetSqlType
     * before being sent to the database.
     *
     * Note that this method may be used to pass datatabase
     * specific abstract data types, by using a Driver specific Java
     * type and using a targetSqlType of java.sql.types.OTHER.
     *
     * @param parameterIndex The first parameter is 1, the second is 2, ...
     * @param x The object containing the input parameter value
     * @param targetSqlType The SQL type (as defined in java.sql.Types) to be
     * sent to the database. The scale argument may further qualify this type.
     * @param scale For java.sql.Types.DECIMAL or java.sql.Types.NUMERIC types
     *          this is the number of digits after the decimal.  For all other
     *          types this value will be ignored,
     */
    void setObject(int parameterIndex, Object x, int targetSqlType, int scale)
            throws SQLException;

    /**
      * This method is like setObject above, but assumes scale of zero.
      */
    void setObject(int parameterIndex, Object x, int targetSqlType) throws SQLEx-
ception;

    /**
     * Set the value of a parameter using an object; use the
     * java.lang equivalent objects for integral values.
     *
     * The JDBC specification specifies a standard mapping from
```

```
         * Java Object types to SQL types.  The given argument java object
         * will be converted to the corresponding SQL type before being
         * sent to the database.
         *
         * Note that this method may be used to pass datatabase
         * specific abstract data types, by using a Driver specific Java
         * type.
         *
         * @param parameterIndex The first parameter is 1, the second is 2, ...
         * @param x The object containing the input parameter value
         */
        void setObject(int parameterIndex, Object x) throws SQLException;

        /**
         * Some prepared statements return multiple results; the execute
         * method handles these complex statements as well as the simpler
         * form of statements handled by executeQuery and executeUpdate.
         *
         */
        boolean execute() throws SQLException;

    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * A ResultSet provides access to a table of data generated by
 * executing a Statement. The table rows are retrieved in
 * sequence. Within a row its column values can be accessed in any
 * order.
 *
 * A ResultSet maintains a cursor pointing to its current row of
 * data.  Initially the cursor is positioned before the first row.
 * The 'next' method moves the cursor to the next row.
 *
 * The getXXX methods retrieve column values for the current
 * row.  You can retrieve values either using the index number of the
 * column, or by using the name of the column.  In general using the
 * column index will be more efficient.  Columns are numbered from 1.
 *
 * For maximum portability, ResultSet columns within each row should be
 * read in left-to-right order and each column should be read only once.
 *
 * For the getXXX methods, the JDBC driver attempts to convert the
 * underlying data to the specified Java type and returns a suitable
 * Java value.  See the JDBC specification for allowable mappings
 * from SQL types to Java types with the ResultSet.getXXX methods.
 *
 * Column names used as input to getXXX methods are case insensitive.
 * When performing a getXXX using a column name if several columns have
 * the same name then the value of the first matching column will be
 * returned.
 *
 * A ResultSet is automatically closed by the Statement that
 * generated it when that Statement is closed, re-executed, or is used
 * to retrieve the next result from a sequence of multiple results.
 *
 * The number, types and properties of a ResultSet's columns are
 * provided by the ResulSetMetaData object returned by the getMetaData
 * method.
 *
 */

public interface ResultSet {

    /**
     * A ResultSet is initially positioned before its first row; the
     * first call to next makes the first row the current row; the
     * second call makes the second row the current row, etc.
     *
     * If an input stream from the previous row is open it is
     * implicitly closed. The ResultSet's warning chain is cleared
     * when a new row is read.
     *
     * @return true if the new current row is valid; false if there
```

```
     * are no more rows
     */
    boolean next() throws SQLException;

    /**
     * In some cases, it is desirable to immediately release a
     * ResultSet's database and JDBC resources instead of waiting for
     * this to happen when it is automatically closed; the close
     * method provides this immediate release.
     *
     * Note: A ResultSet is automatically closed by the
     * Statement that generated it when that Statement is closed,
     * re-executed, or is used to retrieve the next result from a
     * sequence of multiple results. A ResultSet is also automatically
     * closed when it is garbage collected.
     */
    void close() throws SQLException;

    /**
     * A column may have the value of SQL NULL; wasNull reports whether
     * the last column read had this special value.
     * Note that you must first call getXXX on a column to try to read
     * its value and then call wasNull() to find if the value was
     * the SQL NULL.
     *
     * @return true if last column read was SQL NULL
     */
    boolean wasNull() throws SQLException;

    //======================================================================
    // Methods for accessing results by column index
    //======================================================================

    /**
     * Get the value of a column in the current row as a Java String.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is null
     */
    String getString(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java boolean.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is false
     */
    boolean getBoolean(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java byte.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is 0
```

```
     */
    byte getByte(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java short.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is 0
     */
    short getShort(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java int.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is 0
     */
    int getInt(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java long.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is 0
     */
    long getLong(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java float.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is 0
     */
    float getFloat(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java double.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is 0
     */
    double getDouble(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Numeric object.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @param scale the number of digits to the right of the decimal
     * @return the column value; if the value is SQL NULL the result is null
     */
    Numeric getNumeric(int columnIndex, int scale) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java byte array.
```

```
     * The bytes represent the raw values returned by the driver.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is null
     */
    byte[] getBytes(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Date object.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is null
     */
    java.sql.Date getDate(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Time object.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is null
     */
    java.sql.Time getTime(int columnIndex) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Timestamp object.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return the column value; if the value is SQL NULL the result is null
     */
    java.sql.Timestamp getTimestamp(int columnIndex) throws SQLException;

    /**
     * A column value can be retrieved as a stream of ASCII characters
     * and then read in chunks from the stream.  This method is particularly
     * suitable for retrieving large LONGVARCHAR values.  The JDBC driver will
     * do any necessary conversion from the database format into ASCII.
     *
     * Note: All the data in the returned stream must
     * be read prior to getting the value of any other column. The
     * next call to a get method implicitly closes the stream.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return a Java input stream that delivers the database column value
     * as a stream of one byte ASCII characters.  If the value is SQL NULL
     * then the result is null.
     */
    java.io.InputStream getAsciiStream(int columnIndex) throws SQLException;

    /**
     * A column value can be retrieved as a stream of Unicode characters
     * and then read in chunks from the stream.  This method is particularly
     * suitable for retrieving large LONGVARCHAR values.  The JDBC driver will
     * do any necessary conversion from the database format into Unicode.
     *
```

```
     * Note: All the data in the returned stream must
     * be read prior to getting the value of any other column. The
     * next call to a get method implicitly closes the stream.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return a Java input stream that delivers the database column value
     * as a stream of two byte Unicode characters.  If the value is SQL NULL
     * then the result is null.
     */
    java.io.InputStream getUnicodeStream(int columnIndex) throws SQLException;

    /**
     * A column value can be retrieved as a stream of uninterpreted bytes
     * and then read in chunks from the stream.  This method is particularly
     * suitable for retrieving large LONGVARBINARY values.
     *
     * Note: All the data in the returned stream must
     * be read prior to getting the value of any other column. The
     * next call to a get method implicitly closes the stream.
     *
     * @param columnIndex the first column is 1, the second is 2, ...
     * @return a Java input stream that delivers the database column value
     * as a stream of uninterpreted bytes.  If the value is SQL NULL
     * then the result is null.
     */
    java.io.InputStream getBinaryStream(int columnIndex)
        throws SQLException;

    //======================================================================
    // Methods for accessing results by column name
    //======================================================================

    /**
     * Get the value of a column in the current row as a Java String.
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is null
     */
    String getString(String columnName) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java boolean.
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is false
     */
    boolean getBoolean(String columnName) throws SQLException;

    /**
     * Get the value of a column in the current row as a Java byte.
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is 0
     */
```

```
byte getByte(String columnName) throws SQLException;

/**
 * Get the value of a column in the current row as a Java short.
 *
 * @param columnName is the SQL name of the column
 * @return the column value; if the value is SQL NULL the result is 0
 */
short getShort(String columnName) throws SQLException;

/**
 * Get the value of a column in the current row as a Java int.
 *
 * @param columnName is the SQL name of the column
 * @return the column value; if the value is SQL NULL the result is 0
 */
int getInt(String columnName) throws SQLException;

/**
 * Get the value of a column in the current row as a Java long.
 *
 * @param columnName is the SQL name of the column
 * @return the column value; if the value is SQL NULL the result is 0
 */
long getLong(String columnName) throws SQLException;

/**
 * Get the value of a column in the current row as a Java float.
 *
 * @param columnName is the SQL name of the column
 * @return the column value; if the value is SQL NULL the result is 0
 */
float getFloat(String columnName) throws SQLException;

/**
 * Get the value of a column in the current row as a Java double.
 *
 * @param columnName is the SQL name of the column
 * @return the column value; if the value is SQL NULL the result is 0
 */
double getDouble(String columnName) throws SQLException;

/**
 * Get the value of a column in the current row as a java.sql.Numeric object.
 *
 * @param columnName is the SQL name of the column
 * @param scale the number of digits to the right of the decimal
 * @return the column value; if the value is SQL NULL the result is null
 */
Numeric getNumeric(String columnName, int scale) throws SQLException;

/**
 * Get the value of a column in the current row as a Java byte array.
 * The bytes represent the raw values returned by the driver.
```

```
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is null
     */
    byte[] getBytes(String columnName) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Date object.
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is null
     */
    java.sql.Date getDate(String columnName) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Time object.
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is null
     */
    java.sql.Time getTime(String columnName) throws SQLException;

    /**
     * Get the value of a column in the current row as a java.sql.Timestamp object.
     *
     * @param columnName is the SQL name of the column
     * @return the column value; if the value is SQL NULL the result is null
     */
    java.sql.Timestamp getTimestamp(String columnName) throws SQLException;

    /**
     * A column value can be retrieved as a stream of ASCII characters
     * and then read in chunks from the stream.  This method is particularly
     * suitable for retrieving large LONGVARCHAR values.  The JDBC driver will
     * do any necessary conversion from the database format into ASCII.
     *
     * Note: All the data in the returned stream must
     * be read prior to getting the value of any other column. The
     * next call to a get method implicitly closes the stream.
     *
     * @param columnName is the SQL name of the column
     * @return a Java input stream that delivers the database column value
     * as a stream of one byte ASCII characters.  If the value is SQL NULL
     * then the result is null.
     */
    java.io.InputStream getAsciiStream(String columnName) throws SQLException;

    /**
     * A column value can be retrieved as a stream of Unicode characters
     * and then read in chunks from the stream.  This method is particularly
     * suitable for retrieving large LONGVARCHAR values.  The JDBC driver will
     * do any necessary conversion from the database format into Unicode.
     *
     * Note: All the data in the returned stream must
```

```
     * be read prior to getting the value of any other column. The
     * next call to a get method implicitly closes the stream.
     *
     * @param columnName is the SQL name of the column
     * @return a Java input stream that delivers the database column value
     * as a stream of two byte Unicode characters.  If the value is SQL NULL
     * then the result is null.
     */
    java.io.InputStream getUnicodeStream(String columnName) throws SQLException;

    /**
     * A column value can be retrieved as a stream of uninterpreted bytes
     * and then read in chunks from the stream.  This method is particularly
     * suitable for retrieving large LONGVARBINARY values.
     *
     * Note: All the data in the returned stream must
     * be read prior to getting the value of any other column. The
     * next call to a get method implicitly closes the stream.
     *
     * @param columnName is the SQL name of the column
     * @return a Java input stream that delivers the database column value
     * as a stream of uninterpreted bytes.  If the value is SQL NULL
     * then the result is null.
     */
    java.io.InputStream getBinaryStream(String columnName)
        throws SQLException;

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

    /**
     * The first warning reported by calls on this ResultSet is
     * returned. Subsequent ResultSet warnings will be chained to this
     * SQLWarning.
     *
     * The warning chain is automatically cleared each time a new
     * row is read.
     *
     * Note: This warning chain only covers warnings caused
     * by ResultSet methods.  Any warning caused by statement methods
     * (such as reading OUT parameters) will be chained on the
     * Statement object.
     *
     * @return the first SQLWarning or null
     */
    SQLWarning getWarnings() throws SQLException;

    /**
     * After this call getWarnings returns null until a new warning is
     * reported for this ResultSet.
     */
    void clearWarnings() throws SQLException;
```

```
/**
 * Get the name of the SQL cursor used by this ResultSet.
 *
 * In SQL, a result table is retrieved through a cursor that is
 * named. The current row of a result can be updated or deleted
 * using a positioned update/delete statement that references the
 * cursor name.
 *
 * JDBC supports this SQL feature by providing the name of the
 * SQL cursor used by a ResultSet. The current row of a ResultSet
 * is also the current row of this SQL cursor.
 *
 * Note: If positioned update is not supported a
 * SQLException is thrown
 *
 * @return the ResultSet's SQL cursor name
 */
String getCursorName() throws SQLException;

/**
 * The number, types and properties of a ResultSet's columns
 * are provided by the getMetaData method.
 *
 * @return the description of a ResultSet's columns
 */
ResultSetMetaData getMetaData() throws SQLException;

/**
 * Get the value of a column as a Java object.
 *
 * This method will return the value of the given column as a Java
 * object.  The type of the Java object will be default Java Object type
 * corresponding to the column's SQL type, following the mapping
 * specified in the JDBC spec.
 *
 * This method may also be used to read datatabase specific abstract
 * data types.
 *
 * @param columnIndex the first column is 1, the second is 2, ...
 * @return A java.lang.Object holding the column value.
 */
Object getObject(int columnIndex) throws SQLException;

/**
 * Get the value of a parameter as a Java object.
 *
 * This method will return the value of the given column as a Java
 * object.  The type of the Java object will be default Java Object type
 * corresponding to the column's SQL type, following the mapping
 * specified in the JDBC spec.
 *
 * This method may also be used to read datatabase specific abstract
 * data types.
 *
```

```
         * @param columnName is the SQL name of the column
         * @return A java.lang.Object holding the column value.
         */
        Object getObject(String columnName) throws SQLException;

        /**
         * Map a Resultset column name to a ResultSet column index.
         *
         * @param columnName the name of the column
         * @return the column index
         */

        int findColumn(String columnName) throws SQLException;

    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

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

    /**
     * Construct a fully specified SQLException
     *
     * @param reason a description of the exception
     * @param SQLState an XOPEN code identifying the exception
     * @param vendorCode a database vendor specific exception code
     */
    public SQLException(String reason, String SQLState, int vendorCode);

    /**
     * Construct an SQLException without a vendorCode
     *
     * @param reason a description of the exception
     * @param SQLState an XOPEN code identifying the exception
     */
    public SQLException(String reason, String SQLState);

    /**
     * Construct an SQLException with only a reason
     *
     * @param reason a description of the exception
     */
    public SQLException(String reason);

    /**
     * Construct an SQLException with no description
     *
     */
    public SQLException();

    /**
     * Get the SQLState
```

```
        *
        * @return the SQLException's SQLState value
        */
       public String getSQLState();

       /**
        * Get the vendor specific exception code
        *
        * @return the SQLException's vendorCode value
        */
       public int getErrorCode();

       /**
        * Get the exception chained to this one.
        *
        * @return the next SQLException in the chain
        */
       public SQLException getNextException();

       /**
        * Add an SQLException to the end of the chain.
        *
        * @param ex the new end of the SQLException chain
        */
       public synchronized void setNextException(SQLException ex);

   }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * The SQLWarning class provides information on a database access
 * warnings. Warnings are silently chained to the object who's method
 * caused it to be reported.
 *
 */
public class SQLWarning extends SQLException {

    /**
     * Construct a fully specified SQLWarning
     *
     * @param reason a description of the warning
     * @param SQLState an XOPEN code identifying the warning
     * @param vendorCode a database vendor specific warning code
     */
    public SQLWarning(String reason, String SQLstate, int vendorCode);

    /**
     * Construct an SQLWarning without a vendorCode
     *
     * @param reason a description of the warning
     * @param SQLState an XOPEN code identifying the warning
     */
    public SQLWarning(String reason, String SQLstate);

    /**
     * Construct an SQLWarning with only a reason
     *
     * @param reason a description of the warning
     */
    public SQLWarning(String reason);

    /**
     * Construct an SQLWarning with no description
     *
     */
    public SQLWarning();

    /**
     * Get the warning chained to this one
     *
     * @return the next SQLException in the chain
     */
    public SQLWarning getNextWarning();

    /**
     * Add an SQLWarning to the end of the chain.
     *
     * @param w the new end of the SQLException chain
     */
```

```
        public void setNextWarning(SQLWarning w);
    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * A Statement object is used for executing a static SQL statement
 * and obtaining the results produced by it.
 *
 *Only one ResultSet per Statement can be open at any point in
 *time. Therefore, if the reading of one ResultSet is interleaved with
 *the reading of another, each must have been generated by different
 *Statements.
 *
 */
public interface Statement {

    /**
     * Execute a SQL statement that returns a single ResultSet.
     *
     * @param sql typically this is a static SQL SELECT statement
     * @return the table of data produced by the SQL statement
     */
    ResultSet executeQuery(String sql) throws SQLException;

    /**
     * Execute a SQL INSERT, UPDATE or DELETE statement. In addition,
     * SQL statements that return nothing such as SQL DDL statements
     * can be executed.
     *
     * @param sql a SQL INSERT, UPDATE or DELETE statement or a SQL
     * statement that returns nothing
     * @return either the row count for INSERT, UPDATE or DELETE; or 0
     * for SQL statements that return nothing
     */
    int executeUpdate(String sql) throws SQLException;

    /**
     * In many cases, it is desirable to immediately release a
     * Statements's database and JDBC resources instead of waiting for
     * this to happen when it is automatically closed; the close
     * method provides this immediate release.
     *
     * Note: A Statement is automatically closed when it is
     * garbage collected. When a Statement is closed its current
     * ResultSet, if one exists, is also closed.
     */
    void close() throws SQLException;

    /**
     * The maxFieldSize limit (in bytes) is the maximum amount of data
     * returned for any column value; it only applies to BINARY,
     * VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and LONGVARCHAR
     * columns.  If the limit is exceeded, the excess data is silently
     * discarded.
```

```
     *
     * @return the current max column size limit; zero means unlimited
     */
    int getMaxFieldSize() throws SQLException;

    /**
     * The maxFieldSize limit (in bytes) is set to limit the size of
     * data that can be returned for any column value; it only applies
     * to BINARY, VARBINARY, LONGVARBINARY, CHAR, VARCHAR, and
     * LONGVARCHAR fields.  If the limit is exceeded, the excess data
     * is silently discarded.
     *
     * @param max the new max column size limit; zero means unlimited
     */
    void setMaxFieldSize(int max) throws SQLException;

    /**
     * The maxRows limit is the maximum number of rows that a
     * ResultSet can contain.  If the limit is exceeded, the excess
     * rows are silently dropped.
     *
     * @return the current max row limit; zero means unlimited
     */
    int getMaxRows() throws SQLException;

    /**
     * The maxRows limit is set to limit the number of rows that any
     * ResultSet can contain.  If the limit is exceeded, the excess
     * rows are silently dropped.
     *
     * @param max the new max rows limit; zero means unlimited
     */
    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.
     *
     * @param enable true to enable; false to disable
     */
    void setEscapeProcessing(boolean enable) throws SQLException;

    /**
     * The queryTimeout limit is the number of seconds the driver will
     * wait for a Statement to execute. If the limit is exceeded a
     * SQLException is thrown.
     *
     * @return the current query timeout limit in seconds; zero means unlimited
     */
    int getQueryTimeout() throws SQLException;

    /**
     * The queryTimeout limit is the number of seconds the driver will
     * wait for a Statement to execute. If the limit is exceeded a
```

```
     * SQLException is thrown.
     *
    * @param seconds the new query timeout limit in seconds; zero means unlimited
     */
    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;

    /**
     * The first warning reported by calls on this Statement is
     * returned.  A Statment's execute methods clear its SQLWarning
     * chain. Subsequent Statement warnings will be chained to this
     * SQLWarning.
     *
     * The warning chain is automatically cleared each time
     * a statement is (re)executed.
     *
     * Note: If you are processing a ResultSet then any
     * warnings associated with ResultSet reads will be chained on the
     * ResultSet object.
     *
     * @return the first SQLWarning or null
     */
    SQLWarning getWarnings() throws SQLException;

    /**
     * After this call getWarnings returns null until a new warning is
     * reported for this Statement.
     */
    void clearWarnings() throws SQLException;

    /**
     * setCursorname defines the SQL cursor name that will be used by
     * subsequent Statement execute methods. This name can then be
     * used in SQL positioned update/delete statements to identify the
     * current row in the ResultSet generated by this statement.  If
     * the database doesn't support positioned update/delete, this
     * method is a noop.
     *
     * Note: By definition, positioned update/delete
     * execution must be done by a different Statement than the one
     * which generated the ResultSet being used for positioning. Also,
     * cursor names must be unique within a Connection.
     *
     * @param name the new cursor name.
     */
    void setCursorName(String name) throws SQLException;


    /**
```

```
     * Execute a SQL statement that may return multiple results.
     * 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, or unless you're dynamically executing an
     * unknown SQL string.  The "execute", "getMoreResults", "getResultSet"
     * and "getUpdateCount" methods let you navigate through multiple results.
     *
     * The "execute" method executes a SQL statement and indicates the
     * form of the first result.  You can then use getResultSet or
     * getUpdateCount to retrieve the result, and getMoreResults to
     * move to any subsequent result(s).
     *
     * @param sql any SQL statement
     * @return true if the first result is a ResultSet; false if it is an integer
     */
    boolean execute(String sql) throws SQLException;

    /**
     *  getResultSet returns the current result as a ResultSet.  It
     *  should only be called once per result.
     *
     * @return the current result as a ResultSet; null if it is an integer
     */
    ResultSet getResultSet() throws SQLException;

    /**
     *  getUpdateCount returns the current result, which should be an
     *  integer value.  It should only be called once per result.
     *
     * The only way to tell for sure that the result is an update
     *  count is to first test to see if it is a ResultSet. If it is
     *  not a ResultSet it is an update count.
     *
     * @return the current result as an integer; zero if it is a ResultSet
     */
    int getUpdateCount() throws SQLException;

    /**
     * getMoreResults moves to a Statement's next result.  It returns true if
     * this result is a ResultSet.  getMoreResults also implicitly
     * closes any current ResultSet obtained with getResultSet.
     *
     * @return true if the next result is a ResultSet; false if it is an integer
     */
    boolean getMoreResults() throws SQLException;
}
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * 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.
 *
 */
public class Time extends java.util.Date {

    /**
     * Construct a Time
     *
     * @param hour 0 to 23
     * @param minute 0 to 59
     * @param second 0 to 59
     */
    public Time(int hour, int minute, int second);

    /**
     * Convert a formatted string to a Time value
     *
     * @param s time in format “hh:mm:ss”
     * @return corresponding Time
     */
    public static Time valueOf(String s);

    /**
     * Format a time as hh:mm:ss
     *
     * @return a formatted time String
     */
    public String toString ();

}
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

/**
 * This class extends the standard sun.util.date class with nanos.
 * This lets it represent SQL timestamps.
 *
 * Note: The granularity of sub-second timestamp precision may vary
 * between database fields, and stored values will get rounded to the field's
 * internal precision.
 *
 */
public class Timestamp extends java.util.Date {

    /**
     * Construct a Timestamp
     *
     * @param year year-1900
     * @param month 0 to 11
     * @param day 1 to 31
     * @param hour 0 to 23
     * @param minute 0 to 59
     * @param second 0 to 59
     * @param nano 0 to 999,999,999
     */
    public Timestamp(int year, int month, int date,
         int hour, int minute, int second, int nano);

    /**
     * Convert a formatted string to a Timestamp value
     *
     * @param s timestamp in format "yyyy-mm-dd hh:mm:ss.f"
     * @return corresponding Date
     */
    public static Timestamp valueOf(String s);

    /**
     * Format a Timestamp as yyyy-mm-dd hh:mm:ss.f...
     *
     * @return a formatted timestamp String
     */
    public String toString ();

    /**
     * Get the Timestamp's nanos value
     *
     * @return the Timestamp's fractional seconds component
     */
    public int getNanos();

    /**
     * Set the Timestamp's nanos value
     *
```

```
         * @param n the new fractional seconds component
         */
        public void setNanos(int n);

        /**
         * Test Timestamp values for equality
         *
         * @param ts the Timestamp value to compare with
         */
        public boolean equals(Timestamp ts);
    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

package java.sql;

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

    /**
     * OTHER indicates that the SQL type is database specific and
     * gets mapped to a Java object which can be accessed via
     * getObject and setObject.
     */
    public final static int OTHER= 1111;
}
```

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 two 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.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 Prepared-

Statement.setObject method, and a CallableStatement.getObject method. Note that 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 4.

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

Table 4: Mapping from SQL types to Java Object types.

The corresponding default mapping from Java Object types to SQL types is show in Table 5.

Java Object Type	SQL type
String	VARCHAR or LONGVARCHAR
java.sql.Numeric	NUMERIC
Boolean	BIT
Integer	INTEGER
Long	BIGINT
Float	REAL
Double	DOUBLE
byte[]	VARBINARY or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

Table 5: Mapping from Java Object types to SQL types.
Note that the mapping for String will normally be VARCHAR but will turn into LONGVARCHAR if the given value exceeds the drivers limit on VARCHAR values. Similarly for byte[] and VARBINARY and LONGVARBINARY.

Note that it is not possible to send or receive Java input streams using the getObject or setObject methods. You must explicitly use PreparedStatement.setXXXStream or ResultSet.getXXXstream to transfer a value as a stream.

14.2.1 ResultSet.getObject

ResultSet.getObject returns a Java object whose type correspond to the SQL type of the ResultSet column, using the mapping specified in Table 4.

So for example, if you have a ResultSet where the “a” column has SQL type CHAR, and the “b” column has SQL type SMALLINT, here are the types returned by some getObject calls:

```
ResultSet rs = stmt.executeQuery(“SELECT a, b FROM foo”);
while (rs.next()) {
      Object x = rs.getObject(“a”);             // gets a String
      Object y = rs.getObject(“b”);             // gets an Integer
}
```

14.2.2 PreparedStatement.setObject

For PreparedStatement.setObject you can optionally specify a target SQL type. In this case the argument Java Object will first be mapped to its default SQL type (as specified in Table 5) then converted to the specified SQL type (see Table 6), and then sent to the database.

Alternatively you can omit the target SQL type, in which case the given Java Object will simply get mapped to its default SQL type (using Table 5) and then be sent to the database .

14.2.3 CallableStatement.getObject

Before calling CallableStatement.getObject you must first have specified the parameter's SQL type using CallableStatement.registerOutParameter. When you call CallableStatement.getObject the Driver will return a Java Object type corresponding to that SQL type, using the mapping specified Table 4.

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

Table 6: Conversions performed by setObject between Java object types and target SQL types.
An "x" means that the given Java object type may be converted to the given SQL type.
Note that some conversions may fail at runtime if the value presented is invalid.

15 JDBC Metadata Interfaces

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

java.sql.DatabaseMetaData
java.sql.ResultSetMetaData

```
// Copyright (c) 1996 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 show throw a SQLException.
//
// 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?
     *
     * @return true if so
     */
    boolean allProceduresAreCallable() throws SQLException;

    /**
     * Can all the tables returned by getTable be SELECTed by the
     * current user?
     *
     * @return true if so
     */
    boolean allTablesAreSelectable() throws SQLException;

    /**
     * What's the url for this database?
     *
     * @return the url or null if it can't be generated
     */
    String getURL() throws SQLException;

    /**
     * What's our user name as known to the database?
     *
     * @return our database user name
     */
    String getUserName() throws SQLException;

    /**
     * Is the database in read-only mode?
     *
```

```
     * @return true if so
     */
    boolean isReadOnly() throws SQLException;

    /**
     * Are NULL values sorted high?
     *
     * @return true if so
     */
    boolean nullsAreSortedHigh() throws SQLException;

    /**
     * Are NULL values sorted low?
     *
     * @return true if so
     */
    boolean nullsAreSortedLow() throws SQLException;

    /**
     * Are NULL values sorted at the start regardless of sort order?
     *
     * @return true if so
     */
    boolean nullsAreSortedAtStart() throws SQLException;

    /**
     * Are NULL values sorted at the end regardless of sort order?
     *
     * @return true if so
     */
    boolean nullsAreSortedAtEnd() throws SQLException;

    /**
     * What's the name of this database product?
     *
     * @return database product name
     */
    String getDatabaseProductName() throws SQLException;

    /**
     * What's the version of this database product?
     *
     * @return database version
     */
    String getDatabaseProductVersion() throws SQLException;

    /**
     * What's the name of this JDBC driver?
     *
     * @return JDBC driver name
     */
    String getDriverName() throws SQLException;

    /**
```

```
     * What's the version of this JDBC driver?
     *
     * @return JDBC driver version
     */
    String getDriverVersion() throws SQLException;

    /**
     * What's this JDBC driver's major version number?
     *
     * @return JDBC driver major version
     */
    int getDriverMajorVersion();

    /**
     * What's this JDBC driver's minor version number?
     *
     * @return JDBC driver minor version number
     */
    int getDriverMinorVersion();

    /**
     * Does the database store tables in a local file?
     *
     * @return true if so
     */
    boolean usesLocalFiles() throws SQLException;

    /**
     * Does the database use a file for each table?
     *
     * @return true if the database uses a local file for each table
     */
    boolean usesLocalFilePerTable() throws SQLException;

    /**
     * Does the database support mixed case unquoted SQL identifiers?
     *
     * @return true if so
     */
    boolean supportsMixedCaseIdentifiers() throws SQLException;

    /**
     * Does the database store mixed case unquoted SQL identifiers in
     * upper case?
     *
     * @return true if so
     */
    boolean storesUpperCaseIdentifiers() throws SQLException;

    /**
     * Does the database store mixed case unquoted SQL identifiers in
     * lower case?
     *
     * @return true if so
```

```
     */
    boolean storesLowerCaseIdentifiers() throws SQLException;

    /**
     * Does the database store mixed case unquoted SQL identifiers in
     * mixed case?
     *
     * @return true if so
     */
    boolean storesMixedCaseIdentifiers() throws SQLException;

    /**
     * Does the database support mixed case quoted SQL identifiers?
     *
     * A JDBC compliant driver will always return true.
     *
     * @return true if so
     */
    boolean supportsMixedCaseQuotedIdentifiers() throws SQLException;

    /**
     * Does the database store mixed case quoted SQL identifiers in
     * upper case?
     *
     * A JDBC compliant driver will always return true.
     *
     * @return true if so
     */
    boolean storesUpperCaseQuotedIdentifiers() throws SQLException;

    /**
     * Does the database store mixed case quoted SQL identifiers in
     * lower case?
     *
     * A JDBC compliant driver will always return false.
     *
     * @return true if so
     */
    boolean storesLowerCaseQuotedIdentifiers() throws SQLException;

    /**
     * Does the database store mixed case quoted SQL identifiers in
     * mixed case?
     *
     * A JDBC compliant driver will always return false.
     *
     * @return true if so
     */
    boolean storesMixedCaseQuotedIdentifiers() throws SQLException;

    /**
     * What's the string used to quote SQL identifiers?
     * This returns a space " " if identifier quoting isn't supported.
     *
```

```
     * A JDBC compliant driver always uses a double quote character.
     *
     * @return the quoting string
     */
    String getIdentifierQuoteString() throws SQLException;

    /**
     * Get a comma separated list of all a database's SQL keywords
     * that are NOT also SQL92 keywords.
     *
     * @return the list
     */
    String getSQLKeywords() throws SQLException;

    /**
     * Get a comma separated list of math functions.
     *
     * @return the list
     */
    String getNumericFunctions() throws SQLException;

    /**
     * Get a comma separated list of string functions.
     *
     * @return the list
     */
    String getStringFunctions() throws SQLException;

    /**
     * Get a comma separated list of system functions.
     *
     * @return the list
     */
    String getSystemFunctions() throws SQLException;

    /**
     * Get a comma separated list of time and date functions.
     *
     * @return the list
     */
    String getTimeDateFunctions() throws SQLException;

    /**
     * This is the string that can be used to escape '_' or '%' in
     * the string pattern style catalog search parameters.
     *
     * The '_' character represents any single character.
     * The '%' character represents any sequence of zero or
     * more characters.
     * @return the string used to escape wildcard characters
     */
    String getSearchStringEscape() throws SQLException;

    /**
```

```
     * Get all the “extra” characters that can be used in unquoted
     * identifier names (those beyond a-z, 0-9 and _).
     *
     * @return the string containing the extra characters
     */
    String getExtraNameCharacters() throws SQLException;

    // Functions describing which features are supported.

    /**
     * Is “ALTER TABLE” with add column supported?
     *
     * @return true if so
     */
    boolean supportsAlterTableWithAddColumn() throws SQLException;

    /**
     * Is “ALTER TABLE” with drop column supported?
     *
     * @return true if so
     */
    boolean supportsAlterTableWithDropColumn() throws SQLException;

    /**
     * Is column aliasing supported?
     *
     * If so, the SQL AS clause can be used to provide names for
     * computed columns or to provide alias names for columns as
     * required.
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsColumnAliasing() throws SQLException;

    /**
     * Are concatenations between NULL and non-NULL values NULL?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean nullPlusNonNullIsNull() throws SQLException;

    /**
     * Is the CONVERT function between SQL types supported?
     *
     * @return true if so
     */
    boolean supportsConvert() throws SQLException;

    /**
     * Is CONVERT between the given SQL types supported?
```

```
     *
     * @param fromType the type to convert from
     * @param toType the type to convert to
     * @return true if so
     */
    boolean supportsConvert(int fromType, int toType) throws SQLException;

    /**
     * Are table correlation names supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsTableCorrelationNames() throws SQLException;

    /**
     * If table correlation names are supported, are they restricted
     * to be different from the names of the tables?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsDifferentTableCorrelationNames() throws SQLException;

    /**
     * Are expressions in “ORDER BY” lists supported?
     *
     * @return true if so
     */
    boolean supportsExpressionsInOrderBy() throws SQLException;

    /**
     * Can an “ORDER BY” clause use columns not in the SELECT?
     *
     * @return true if so
     */
    boolean supportsOrderByUnrelated() throws SQLException;

    /**
     * Is some form of “GROUP BY” clause supported?
     *
     * @return true if so
     */
    boolean supportsGroupBy() throws SQLException;

    /**
     * Can a “GROUP BY” clause use columns not in the SELECT?
     *
     * @return true if so
     */
    boolean supportsGroupByUnrelated() throws SQLException;
```

```
/**
 * Can a “GROUP BY” clause add columns not in the SELECT
 * provided it specifies all the columns in the SELECT?
 *
 * @return true if so
 */
boolean supportsGroupByBeyondSelect() throws SQLException;

/**
 * Is the escape character in “LIKE” clauses supported?
 *
 * A JDBC compliant driver always returns true.
 *
 * @return true if so
 */
boolean supportsLikeEscapeClause() throws SQLException;

/**
 * Are multiple ResultSets from a single execute supported?
 *
 * @return true if so
 */
boolean supportsMultipleResultSets() throws SQLException;

/**
 * Can we have multiple transactions open at once (on different
 * connections)?
 *
 * @return true if so
 */
boolean supportsMultipleTransactions() throws SQLException;

/**
 * Can columns be defined as non-nullable?
 *
 * A JDBC compliant driver always returns true.
 *
 * @return true if so
 */
boolean supportsNonNullableColumns() throws SQLException;

/**
 * Is the ODBC Minimum SQL grammar supported?
 *
 * All JDBC compliant drivers must return true.
 *
 * @return true if so
 */
boolean supportsMinimumSQLGrammar() throws SQLException;

/**
 * Is the ODBC Core SQL grammar supported?
 *
 * @return true if so
```

```
     */
    boolean supportsCoreSQLGrammar() throws SQLException;

    /**
     * Is the ODBC Extended SQL grammar supported?
     *
     * @return true if so
     */
    boolean supportsExtendedSQLGrammar() throws SQLException;

    /**
     * Is the ANSI92 entry level SQL grammar supported?
     *
     * All JDBC compliant drivers must return true.
     *
     * @return true if so
     */
    boolean supportsANSI92EntryLevelSQL() throws SQLException;

    /**
     * Is the ANSI92 intermediate SQL grammar supported?
     *
     * @return true if so
     */
    boolean supportsANSI92IntermediateSQL() throws SQLException;

    /**
     * Is the ANSI92 full SQL grammar supported?
     *
     * @return true if so
     */
    boolean supportsANSI92FullSQL() throws SQLException;

    /**
     * Is the SQL Integrity Enhancement Facility supported?
     *
     * @return true if so
     */
    boolean supportsIntegrityEnhancementFacility() throws SQLException;

    /**
     * Is some form of outer join supported?
     *
     * @return true if so
     */
    boolean supportsOuterJoins() throws SQLException;

    /**
     * Are full nested outer joins supported?
     *
     * @return true if so
     */
    boolean supportsFullOuterJoins() throws SQLException;
```

```
/**
 * Is there limited support for outer joins?  (This will be true
 * if supportFullOuterJoins is true.)
 *
 * @return true if so
 */
boolean supportsLimitedOuterJoins() throws SQLException;

/**
 * What's the database vendor's preferred term for "schema"?
 *
 * @return the vendor term
 */
String getSchemaTerm() throws SQLException;

/**
 * What's the database vendor's preferred term for "procedure"?
 *
 * @return the vendor term
 */
String getProcedureTerm() throws SQLException;

/**
 * What's the database vendor's preferred term for "catalog"?
 *
 * @return the vendor term
 */
String getCatalogTerm() throws SQLException;

/**
 * Does a catalog appear at the start of a qualified table name?
 * (Otherwise it appears at the end)
 *
 * @return true if it appears at the start
 */
boolean isCatalogAtStart() throws SQLException;

/**
 * What's the separator between catalog and table name?
 *
 * @return the separator string
 */
String getCatalogSeparator() throws SQLException;

/**
 * Can a schema name be used in a data manipulation statement?
 *
 * @return true if so
 */
boolean supportsSchemasInDataManipulation() throws SQLException;

/**
 * Can a schema name be used in a procedure call statement?
 *
```

```
     * @return true if so
     */
    boolean supportsSchemasInProcedureCalls() throws SQLException;

    /**
     * Can a schema name be used in a table definition statement?
     *
     * @return true if so
     */
    boolean supportsSchemasInTableDefinitions() throws SQLException;

    /**
     * Can a schema name be used in an index definition statement?
     *
     * @return true if so
     */
    boolean supportsSchemasInIndexDefinitions() throws SQLException;

    /**
     * Can a schema name be used in a privilege definition statement?
     *
     * @return true if so
     */
    boolean supportsSchemasInPrivilegeDefinitions() throws SQLException;

    /**
     * Can a catalog name be used in a data manipulation statement?
     *
     * @return true if so
     */
    boolean supportsCatalogsInDataManipulation() throws SQLException;

    /**
     * Can a catalog name be used in a procedure call statement?
     *
     * @return true if so
     */
    boolean supportsCatalogsInProcedureCalls() throws SQLException;

    /**
     * Can a catalog name be used in a table definition statement?
     *
     * @return true if so
     */
    boolean supportsCatalogsInTableDefinitions() throws SQLException;

    /**
     * Can a catalog name be used in a index definition statement?
     *
     * @return true if so
     */
    boolean supportsCatalogsInIndexDefinitions() throws SQLException;

    /**
```

```
     * Can a catalog name be used in a privilege definition statement?
     *
     * @return true if so
     */
    boolean supportsCatalogsInPrivilegeDefinitions() throws SQLException;

    /**
     * Is positioned DELETE supported?
     *
     * @return true if so
     */
    boolean supportsPositionedDelete() throws SQLException;

    /**
     * Is positioned UPDATE supported?
     *
     * @return true if so
     */
    boolean supportsPositionedUpdate() throws SQLException;

    /**
     * Is SELECT for UPDATE supported?
     *
     * @return true if so
     */
    boolean supportsSelectForUpdate() throws SQLException;

    /**
     * Are stored procedure calls using the stored procedure escape
     * syntax supported?
     *
     * @return true if so
     */
    boolean supportsStoredProcedures() throws SQLException;

    /**
     * Are subqueries in comparison expressions supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsSubqueriesInComparisons() throws SQLException;

    /**
     * Are subqueries in exists expressions supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsSubqueriesInExists() throws SQLException;

    /**
```

```
     * Are subqueries in “in” statements supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsSubqueriesInIns() throws SQLException;

    /**
     * Are subqueries in quantified expressions supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsSubqueriesInQuantifieds() throws SQLException;

    /**
     * Are correlated subqueries supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsCorrelatedSubqueries() throws SQLException;

    /**
     * Is SQL UNION supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsUnion() throws SQLException;

    /**
     * Is SQL UNION ALL supported?
     *
     * A JDBC compliant driver always returns true.
     *
     * @return true if so
     */
    boolean supportsUnionAll() throws SQLException;

    /**
     * Can cursors remain open across commits?
     *
     * @return true if so
     */
    boolean supportsOpenCursorsAcrossCommit() throws SQLException;

    /**
     * Can cursors remain open across rollbacks?
     *
```

```
     * @return true if so
     */
    boolean supportsOpenCursorsAcrossRollback() throws SQLException;

    /**
     * Can statements remain open across commits?
     *
     * @return true if so
     */
    boolean supportsOpenStatementsAcrossCommit() throws SQLException;

    /**
     * Can statements remain open across rollbacks?
     *
     * @return true if so
     */
    boolean supportsOpenStatementsAcrossRollback() throws SQLException;



    // The following group of methods exposes various limitations
    // based on the target database with the current driver.
    // Unless otherwise 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?
     *
     * @return max literal length
     */
    int getMaxBinaryLiteralLength() throws SQLException;

    /**
     * What's the max length for a character literal?
     *
     * @return max literal length
     */
    int getMaxCharLiteralLength() throws SQLException;

    /**
     * What's the limit on column name length?
     *
     * @return max literal length
     */
    int getMaxColumnNameLength() throws SQLException;

    /**
     * What's the maximum number of columns in a "GROUP BY" clause?
     *
     * @return max number of columns
     */
    int getMaxColumnsInGroupBy() throws SQLException;

    /**
```

```
     * What's the maximum number of columns allowed in an index?
     *
     * @return max columns
     */
    int getMaxColumnsInIndex() throws SQLException;

    /**
     * What's the maximum number of columns in an "ORDER BY" clause?
     *
     * @return max columns
     */
    int getMaxColumnsInOrderBy() throws SQLException;

    /**
     * What's the maximum number of columns in a "SELECT" list?
     *
     * @return max columns
     */
    int getMaxColumnsInSelect() throws SQLException;

    /**
     * What's maximum number of columns in a table?
     *
     * @return max columns
     */
    int getMaxColumnsInTable() throws SQLException;

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

    /**
     * What's the maximum cursor name length?
     *
     * @return max cursor name length in bytes
     */
    int getMaxCursorNameLength() throws SQLException;

    /**
     * What's the maximum length of an index (in bytes)?
     *
     * @return max index length in bytes
     */
    int getMaxIndexLength() throws SQLException;

    /**
     * What's the maximum length allowed for a schema name?
     *
     * @return max name length in bytes
     */
    int getMaxSchemaNameLength() throws SQLException;
```

```
/**
 * What's the maximum length of a procedure name?
 *
 * @return max name length in bytes
 */
int getMaxProcedureNameLength() throws SQLException;

/**
 * What's the maximum length of a catalog name?
 *
 * @return max name length in bytes
 */
int getMaxCatalogNameLength() throws SQLException;

/**
 * What's the maximum length of a single row?
 *
 * @return max row size in bytes
 */
int getMaxRowSize() throws SQLException;

/**
 * Did getMaxRowSize() include LONGVARCHAR and LONGVARBINARY
 * blobs?
 *
 * @return true if so
 */
boolean doesMaxRowSizeIncludeBlobs() throws SQLException;

/**
 * What's the maximum length of a SQL statement?
 *
 * @return max length in bytes
 */
int getMaxStatementLength() throws SQLException;

/**
 * How many active statements can we have open at one time to this
 * database?
 *
 * @return the maximum
 */
int getMaxStatements() throws SQLException;

/**
 * What's the maximum length of a table name?
 *
 * @return max name length in bytes
 */
int getMaxTableNameLength() throws SQLException;

/**
 * What's the maximum number of tables in a SELECT?
```

```
     *
     * @return the maximum
     */
    int getMaxTablesInSelect() throws SQLException;

    /**
     * What's the maximum length of a user name?
     *
     * @return max name length  in bytes
     */
    int getMaxUserNameLength() throws SQLException;

    /**
     * What's the database's default transaction isolation level?  The
     * values are defined in java.sql.Connection.
     *
     * @return the default isolation level
     */
    int getDefaultTransactionIsolation() throws SQLException;

    /**
     * Are transactions supported? If not, commit is a noop and the
     * isolation level is TRANSACTION_NONE.
     *
     * @return true if transactions are supported
     */
    boolean supportsTransactions() throws SQLException;

    /**
     * Does the database support the given transaction isolation level?
     *
     * @param level the values are defined in java.sql.Connection
     * @return true if so
     */
    boolean supportsTransactionIsolationLevel(int level)
                        throws SQLException;

    /**
     * Are both data definition and data manipulation statements
     * within a transaction supported?
     *
     * @return true if so
     */
    boolean supportsDataDefinitionAndDataManipulationTransactions()
                         throws SQLException;
    /**
     * Are only data manipulation statements within a transaction
     * supported?
     *
     * @return true if so
     */
    boolean supportsDataManipulationTransactionsOnly()
                        throws SQLException;
    /**
```

```
     * Does a data definition statement within a transaction force the
     * transaction to commit?
     *
     * @return true if so
     */
    boolean dataDefinitionCausesTransactionCommit()
                        throws SQLException;
    /**
     * Is a data definition statement within a transaction ignored?
     *
     * @return true if so
     */
    boolean dataDefinitionIgnoredInTransactions()
                        throws SQLException;

    /**
     * Get a description of stored procedures available in a
     * catalog.
     *
     * Only procedure descriptions matching the schema and
     * procedure name criteria are returned.  They are ordered by
     * PROCEDURE_SCHEM, and PROCEDURE_NAME.
     *
     * Each procedure description has the the following columns:
     *  <OL>
     *PROCEDURE_CAT String => procedure catalog (may be null)
     *PROCEDURE_SCHEM String => procedure schema (may be null)
     *PROCEDURE_NAME String => procedure name
     *   reserved for future use
     *REMARKS String => explanatory comment on the procedure
     *PROCEDURE_TYPE short => kind of procedure:
     *
     *       procedureResultUnknown - May return a result
     *       procedureNoResult - Does not return a result
     *       procedureReturnsResult - Returns a result
     *
     *  </OL>
     *
     * @param catalog a catalog name; "" retrieves those without a catalog
     * @param schemaPattern a schema name pattern; "" retrieves those
     * without a schema
     * @param procedureNamePattern a procedure name pattern
     * @return ResultSet each row is a procedure description
     */
    ResultSet getProcedures(String catalog, String schemaPattern,
        String procedureNamePattern) throws SQLException;

    /**
     * PROCEDURE_TYPE - May return a result.
     */
    int procedureResultUnknown= 0;
    /**
     * PROCEDURE_TYPE - Does not return a result.
     */
```

```
int procedureNoResult= 1;
/**
 * PROCEDURE_TYPE - Returns a result.
 */
int procedureReturnsResult= 2;

/**
 * Get a description of a catalog's stored procedure parameters
 * and result columns.
 *
 * Only descriptions matching the schema, procedure and
 * parameter name criteria are returned.  They are ordered by
 * PROCEDURE_SCHEM and PROCEDURE_NAME. Within this, the return value,
 * if any, is first. Next are the parameter descriptions in call
 * order. The column descriptions follow in column number order.
 *
 * Each row in the ResultSet is a parameter desription or
 * column description with the following fields:
 *  <OL>
 *PROCEDURE_CAT String => procedure catalog (may be null)
 *PROCEDURE_SCHEM String => procedure schema (may be null)
 *PROCEDURE_NAME String => procedure name
 *COLUMN_NAME String => column/parameter name
 *COLUMN_TYPE Short => kind of column/parameter:
 *
 *       procedureColumnUnknown - nobody knows
 *       procedureColumnIn - IN parameter
 *       procedureColumnInOut - INOUT parameter
 *       procedureColumnOut - OUT parameter
 *       procedureColumnReturn - procedure return value
 *       procedureColumnResult - result column in ResultSet
 *
 *  DATA_TYPE short => SQL type from java.sql.Types
 *TYPE_NAME String => SQL type name
 *PRECISION int => precision
 *LENGTH int => length in bytes of data
 *SCALE short => scale
 *RADIX short => radix
 *NULLABLE short => can it contain NULL?
 *
 *       procedureNoNulls - does not allow NULL values
 *       procedureNullable - allows NULL values
 *       procedureNullableUnknown - nullability unknown
 *
 *REMARKS String => comment describing parameter/column
 *  </OL>
 *
 * Note: Some databases may not return the column
 * descriptions for a procedure. Additional columns beyond
 * REMARKS can be defined by the database.
 *
 * @param catalog a catalog name; "" retrieves those without a catalog
 * @param schemaPattern a schema name pattern; "" retrieves those
 * without a schema
```

```
     * @param procedureNamePattern a procedure name pattern
     * @param columnNamePattern a column name pattern
     * @return ResultSet each row is a stored procedure parameter or
     *      column description
     */
    ResultSet getProcedureColumns(String catalog,
        String schemaPattern,
        String procedureNamePattern,
        String columnNamePattern) throws SQLException;

    /**
     * COLUMN_TYPE - nobody knows.
     */
    int procedureColumnUnknown = 0;

    /**
     * COLUMN_TYPE - IN parameter.
     */
    int procedureColumnIn = 1;

    /**
     * COLUMN_TYPE - INOUT parameter.
     */
    int procedureColumnInOut = 2;

    /**
     * COLUMN_TYPE - OUT parameter.
     */
    int procedureColumnOut = 4;
    /**
     * COLUMN_TYPE - procedure return value.
     */
    int procedureColumnReturn = 5;

    /**
     * COLUMN_TYPE - result column in ResultSet.
     */
    int procedureColumnResult = 3;

    /**
     * TYPE NULLABLE - does not allow NULL values.
     */
    int procedureNoNulls = 0;

    /**
     * TYPE NULLABLE - allows NULL values.
     */
    int procedureNullable = 1;

    /**
     * TYPE NULLABLE - nullability unknown.
     */
    int procedureNullableUnknown = 2;
```

```
/**
 * Get a description of tables available in a catalog.
 *
 * Only table descriptions matching the catalog, schema, table
 * name and type criteria are returned.  They are ordered by
 * TABLE_TYPE, TABLE_SCHEM and TABLE_NAME.
 *
 * Each table description has the following columns:
 *  <OL>
 *TABLE_CAT String => table catalog (may be null)
 *TABLE_SCHEM String => table schema (may be null)
 *TABLE_NAME String => table name
 *TABLE_TYPE String => table type.  Typical types are “TABLE”,
 *      “VIEW”,“SYSTEM TABLE”, “GLOBAL TEMPORARY”,
 *      “LOCAL TEMPORARY”, “ALIAS”, “SYNONYM”.
 *REMARKS String => explanatory comment on the table
 *  </OL>
 *
 * Note: Some databases may not return information for
 * all tables.
 *
 * @param catalog a catalog name; “” retrieves those without a catalog
 * @param schemaPattern a schema name pattern; “” retrieves those
 * without a schema
 * @param tableNamePattern a table name pattern
 * @param types a list of table types to include; null returns all types
 * @return ResultSet each row is a table description
 */
ResultSet getTables(String catalog, String schemaPattern,
String tableNamePattern, String types[]) throws SQLException;

/**
 * Get the schema names available in this database.  The results
 * are ordered by schema name.
 *
 * The schema column is:
 *  <OL>
 *TABLE_SCHEM String => schema name
 *  </OL>
 *
 * @return ResultSet each row has a single String column that is a
 * schema name
 */
ResultSet getSchemas() throws SQLException;

/**
 * Get the catalog names available in this database.  The results
 * are ordered by catalog name.
 *
 * The catalog column is:
 *  <OL>
 *TABLE_CAT String => catalog name
 *  </OL>
 *
```

```
     * @return ResultSet each row has a single String column that is a
     * catalog name
     */
    ResultSet getCatalogs() throws SQLException;

    /**
     * Get the table types available in this database.  The results
     * are ordered by table type.
     *
     * The table type is:
     *  <OL>
     *TABLE_TYPE String => table type.  Typical types are “TABLE”,
     *      “VIEW”,“SYSTEM TABLE”, “GLOBAL TEMPORARY”,
     *      “LOCAL TEMPORARY”, “ALIAS”, “SYNONYM”.
     *  </OL>
     *
     * @return ResultSet each row has a single String column that is a
     * table type
     */
    ResultSet getTableTypes() throws SQLException;

    /**
     * Get a description of table columns available in a catalog.
     *
     * Only column descriptions matching the catalog, schema, table
     * and column name criteria are returned.  They are ordered by
     * TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.
     *
     * Each column description has the following columns:
     *  <OL>
     *TABLE_CAT String => table catalog (may be null)
     *TABLE_SCHEM String => table schema (may be null)
     *TABLE_NAME String => table name
     *COLUMN_NAME String => column name
     *DATA_TYPE short => SQL type from java.sql.Types
     *TYPE_NAME String => Data source dependent type name
     *COLUMN_SIZE int => column size.  For char or date
     *    types this is the maximum number of characters, for numeric or
     *    decimal types this is precision.
     *BUFFER_LENGTH is not used.
     *DECIMAL_DIGITS int => the number of fractional digits
     *NUM_PREC_RADIX int => Radix (typically either 10 or 2)
     *NULLABLE int => is NULL allowed?
     *
     *       columnNoNulls - might not allow NULL values
     *       columnNullable - definitely allows NULL values
     *       columnNullableUnknown - nullability unknown
     *
     *REMARKS String => comment describing column (may be null)
     * COLUMN_DEF String => default value (may be null)
     *SQL_DATA_TYPE int => unused
     *SQL_DATETIME_SUB int => unused
     *CHAR_OCTET_LENGTH int => for char types the
     *       maximum number of bytes in the column
```

```
 *ORDINAL_POSITION int=> index of column in table
 *      (starting at 1)
 *IS_NULLABLE String => "NO" means column definitely
 *      does not allow NULL values; "YES" means the column might
 *      allow NULL values.  An empty string means nobody knows.
 *  </OL>
 *
 * @param catalog a catalog name; "" retrieves those without a catalog
 * @param schemaPattern a schema name pattern; "" retrieves those
 * without a schema
 * @param tableNamePattern a table name pattern
 * @param columnNamePattern a column name pattern
 * @return ResultSet each row is a column description
 */
ResultSet getColumns(String catalog, String schemaPattern,
String tableNamePattern, String columnNamePattern)
            throws SQLException;
/**
 * COLUMN NULLABLE - might not allow NULL values.
 */
int columnNoNulls = 0;

/**
 * COLUMN NULLABLE - definitely allows NULL values.
 */
int columnNullable = 1;

/**
 * COLUMN NULLABLE - nullability unknown.
 */
int columnNullableUnknown = 2;

/**
 * Get a description of the access rights for a table's columns.
 *
 * Only privileges matching the column name criteria are
 * returned.  They are ordered by COLUMN_NAME and PRIVILEGE.
 *
 * Each privilige description has the following columns:
 *  <OL>
 *TABLE_CAT String => table catalog (may be null)
 *TABLE_SCHEM String => table schema (may be null)
 *TABLE_NAME String => table name
 *COLUMN_NAME String => column name
 *GRANTOR => grantor of access (may be null)
 *GRANTEE String => grantee of access
 *PRIVILEGE String => name of access (SELECT,
 *      INSERT, UPDATE, REFRENCES, ...)
 *IS_GRANTABLE String => "YES" if grantee is permitted
 *      to grant to others; "NO" if not; null if unknown
 *  </OL>
 *
 * @param catalog a catalog name; "" retrieves those without a catalog
 * @param schema a schema name; "" retrieves those without a schema
```

```
     * @param table a table name
     * @param columnNamePattern a column name pattern
     * @return ResultSet each row is a column privilege description
     */
    ResultSet getColumnPrivileges(String catalog, String schema,
    String table, String columnNamePattern) throws SQLException;

    /**
     * Get a description of the access rights for each table available
     * in a catalog.
     *
     * Only privileges matching the schema and table name
     * criteria are returned.  They are ordered by TABLE_SCHEM,
     * TABLE_NAME, and PRIVILEGE.
     *
     * Each privilige description has the following columns:
     *  <OL>
     *TABLE_CAT String => table catalog (may be null)
     *TABLE_SCHEM String => table schema (may be null)
     *TABLE_NAME String => table name
     *COLUMN_NAME String => column name
     *GRANTOR => grantor of access (may be null)
     *GRANTEE String => grantee of access
     *PRIVILEGE String => name of access (SELECT,
     *      INSERT, UPDATE, REFRENCES, ...)
     *IS_GRANTABLE String => “YES” if grantee is permitted
     *      to grant to others; “NO” if not; null if unknown
     *  </OL>
     *
     * @param catalog a catalog name; “” retrieves those without a catalog
     * @param schemaPattern a schema name pattern; “” retrieves those
     * without a schema
     * @param tableNamePattern a table name pattern
     * @return ResultSet each row is a table privilege description
     */
    ResultSet getTablePrivileges(String catalog, String schemaPattern,
            String tableNamePattern) throws SQLException;

    /**
     * Get a description of a table’s optimal set of columns that
     * uniquely identifies a row. They are ordered by SCOPE.
     *
     * Each column description has the following columns:
     *  <OL>
     *SCOPE short => actual scope of result
     *
     *       bestRowTemporary - very temporary, while using row
     *       bestRowTransaction - valid for remainder of current transaction
     *       bestRowSession - valid for remainder of current session
     *
     *COLUMN_NAME String => column name
     *DATA_TYPE short => SQL data type from java.sql.Types
     *TYPE_NAME String => Data source dependent type name
     *COLUMN_SIZE int => precision
```

```
 *BUFFER_LENGTH int => not used
 *DECIMAL_DIGITS short => scale
 *PSEUDO_COLUMN short => is this a pseudo column
 *      like an Oracle ROWID
 *
 *       bestRowUnknown - may or may not be pseudo column
 *       bestRowNotPseudo - is NOT a pseudo column
 *       bestRowPseudo - is a pseudo column
 *
 *  </OL>
 *
 * @param catalog a catalog name; "" retrieves those without a catalog
 * @param schema a schema name; "" retrieves those without a schema
 * @param table a table name
 * @param scope the scope of interest; use same values as SCOPE
 * @param nullable include columns that are nullable?
 * @return ResultSet each row is a column description
 */
ResultSet getBestRowIdentifier(String catalog, String schema,
String table, int scope, boolean nullable) throws SQLException;

/**
 * BEST ROW SCOPE - very temporary, while using row.
 */
int bestRowTemporary   = 0;

/**
 * BEST ROW SCOPE - valid for remainder of current transaction.
 */
int bestRowTransaction = 1;

/**
 * BEST ROW SCOPE - valid for remainder of current session.
 */
int bestRowSession     = 2;

/**
 * BEST ROW PSEUDO_COLUMN - may or may not be pseudo column.
 */
int bestRowUnknown= 0;

/**
 * BEST ROW PSEUDO_COLUMN - is NOT a pseudo column.
 */
int bestRowNotPseudo= 1;

/**
 * BEST ROW PSEUDO_COLUMN - is a pseudo column.
 */
int bestRowPseudo= 2;

/**
 * Get a description of a table's columns that are automatically
 * updated when any value in a row is updated.  They are
```

```
     * unordered.
     *
     * Each column description has the following columns:
     *  <OL>
     *SCOPE short => is not used
     *COLUMN_NAME String => column name
     *DATA_TYPE short => SQL data type from java.sql.Types
     *TYPE_NAME String => Data source dependent type name
     *COLUMN_SIZE int => precision
     *BUFFER_LENGTH int => length of column value in bytes
     *DECIMAL_DIGITS short => scale
     *PSEUDO_COLUMN short => is this a pseudo column
     *      like an Oracle ROWID
     *
     *       versionColumnUnknown - may or may not be pseudo column
     *       versionColumnNotPseudo - is NOT a pseudo column
     *       versionColumnPseudo - is a pseudo column
     *
     *  </OL>
     *
     * @param catalog a catalog name; "" retrieves those without a catalog
     * @param schema a schema name; "" retrieves those without a schema
     * @param table a table name
     * @return ResultSet each row is a column description
     */
    ResultSet getVersionColumns(String catalog, String schema,
            String table) throws SQLException;

    /**
     * VERSION COLUMNS PSEUDO_COLUMN - may or may not be pseudo column.
     */
    int versionColumnUnknown= 0;

    /**
     *  VERSION COLUMNS PSEUDO_COLUMN - is NOT a pseudo column.
     */
    int versionColumnNotPseudo= 1;

    /**
     *  VERSION COLUMNS PSEUDO_COLUMN - is a pseudo column.
     */
    int versionColumnPseudo= 2;

    /**
     * Get a description of a table's primary key columns.  They
     * are ordered by COLUMN_NAME.
     *
     * Each column description has the following columns:
     *  <OL>
     *TABLE_CAT String => table catalog (may be null)
     *TABLE_SCHEM String => table schema (may be null)
     *TABLE_NAME String => table name
     *COLUMN_NAME String => column name
     *KEY_SEQ short => sequence number within primary key
```

```
     *PK_NAME String => primary key name (may be null)
     *  </OL>
     *
     * @param catalog a catalog name; "" retrieves those without a catalog
     * @param schema a schema name pattern; "" retrieves those
     * without a schema
     * @param table a table name
     * @return ResultSet each row is a primary key column description
     */
    ResultSet getPrimaryKeys(String catalog, String schema,
            String table) throws SQLException;

    /**
     * Get a description of the primary key columns that are
     * referenced by a table's foreign key columns (the primary keys
     * imported by a table).  They are ordered by PKTABLE_CAT,
     * PKTABLE_SCHEM, PKTABLE_NAME, and KEY_SEQ.
     *
     * Each primary key column description has the following columns:
     *  <OL>
     *PKTABLE_CAT String => primary key table catalog
     *      being imported (may be null)
     *PKTABLE_SCHEM String => primary key table schema
     *      being imported (may be null)
     *PKTABLE_NAME String => primary key table name
     *      being imported
     *PKCOLUMN_NAME String => primary key column name
     *      being imported
     *FKTABLE_CAT String => foreign key table catalog (may be null)
     *FKTABLE_SCHEM String => foreign key table schema (may be null)
     *FKTABLE_NAME String => foreign key table name
     *FKCOLUMN_NAME String => foreign key column name
     *KEY_SEQ short => sequence number within foreign key
     *UPDATE_RULE short => What happens to
     *       foreign key when primary is updated:
     *
     *       importedKeyCascade - change imported key to agree
     *               with primary key update
     *       importedKeyRestrict - do not allow update of primary
     *               key if it has been imported
     *       importedKeySetNull - change imported key to NULL if
     *               its primary key has been updated
     *
     *DELETE_RULE short => What happens to
     *      the foreign key when primary is deleted.
     *
     *       importedKeyCascade - delete rows that import a deleted key
     *       importedKeyRestrict - do not allow delete of primary
     *               key if it has been imported
     *       importedKeySetNull - change imported key to NULL if
     *               its primary key has been deleted
     *
     *FK_NAME String => foreign key name (may be null)
     *PK_NAME String => primary key name (may be null)
```

```
     *  </OL>
     *
     * @param catalog a catalog name; "" retrieves those without a catalog
     * @param schema a schema name pattern; "" retrieves those
     * without a schema
     * @param table a table name
     * @return ResultSet each row is a primary key column description
     */
    ResultSet getImportedKeys(String catalog, String schema,
            String table) throws SQLException;

    /**
     * IMPORT KEY UPDATE_RULE and DELETE_RULE - for update, change
     * imported key to agree with primary key update; for delete,
     * delete rows that import a deleted key.
     */
    int importedKeyCascade= 0;

    /**
     * IMPORT KEY UPDATE_RULE and DELETE_RULE - do not allow update or
     * delete of primary key if it has been imported.
     */
    int importedKeyRestrict = 1;

    /**
     * IMPORT KEY UPDATE_RULE and DELETE_RULE - change imported key to
     * NULL if its primary key has been updated or deleted.
     */
    int importedKeySetNull  = 2;

    /**
     * Get a description of a foreign key columns that reference a
     * table's primary key columns (the foreign keys exported by a
     * table).  They are ordered by FKTABLE_CAT, FKTABLE_SCHEM,
     * FKTABLE_NAME, and KEY_SEQ.
     *
     * Each foreign key column description has the following columns:
     *  <OL>
     *PKTABLE_CAT String => primary key table catalog (may be null)
     *PKTABLE_SCHEM String => primary key table schema (may be null)
     *PKTABLE_NAME String => primary key table name
     *PKCOLUMN_NAME String => primary key column name
     *FKTABLE_CAT String => foreign key table catalog (may be null)
     *      being exported (may be null)
     *FKTABLE_SCHEM String => foreign key table schema (may be null)
     *      being exported (may be null)
     *FKTABLE_NAME String => foreign key table name
     *      being exported
     *FKCOLUMN_NAME String => foreign key column name
     *      being exported
     *KEY_SEQ short => sequence number within foreign key
     *UPDATE_RULE short => What happens to
     *       foreign key when primary is updated:
     *
```

```
     *       importedKeyCascade - change imported key to agree
     *               with primary key update
     *       importedKeyRestrict - do not allow update of primary
     *               key if it has been imported
     *       importedKeySetNull - change imported key to NULL if
     *               its primary key has been updated
     *
     *DELETE_RULE short => What happens to
     *      the foreign key when primary is deleted.
     *
     *       importedKeyCascade - delete rows that import a deleted key
     *       importedKeyRestrict - do not allow delete of primary
     *               key if it has been imported
     *       importedKeySetNull - change imported key to NULL if
     *               its primary key has been deleted
     *
     *FK_NAME String => foreign key identifier (may be null)
     *PK_NAME String => primary key identifier (may be null)
     *  </OL>
     *
     * @param catalog a catalog name; "" retrieves those without a catalog
     * @param schema a schema name pattern; "" retrieves those
     * without a schema
     * @param table a table name
     * @return ResultSet each row is a foreign key column description
     */
    ResultSet getExportedKeys(String catalog, String schema,
            String table) throws SQLException;

    /**
     * Get a description of the foreign key columns in the foreign key
     * table that reference the primary key columns of the primary key
     * table (describe how one table imports another's key.) This
     * should normally return a single foreign key/primary key pair
     * (most tables only import a foreign key from a table once.)  They
     * are ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and
     * KEY_SEQ.
     *
     * Each foreign key column description has the following columns:
     *  <OL>
     *PKTABLE_CAT String => primary key table catalog (may be null)
     *PKTABLE_SCHEM String => primary key table schema (may be null)
     *PKTABLE_NAME String => primary key table name
     *PKCOLUMN_NAME String => primary key column name
     *FKTABLE_CAT String => foreign key table catalog (may be null)
     *      being exported (may be null)
     *FKTABLE_SCHEM String => foreign key table schema (may be null)
     *      being exported (may be null)
     *FKTABLE_NAME String => foreign key table name
     *      being exported
     *FKCOLUMN_NAME String => foreign key column name
     *      being exported
     *KEY_SEQ short => sequence number within foreign key
     *UPDATE_RULE short => What happens to
```

```
     *       foreign key when primary is updated:
     *
     *       importedKeyCascade - change imported key to agree
     *               with primary key update
     *       importedKeyRestrict - do not allow update of primary
     *               key if it has been imported
     *       importedKeySetNull - change imported key to NULL if
     *               its primary key has been updated
     *
     *DELETE_RULE short => What happens to
     *      the foreign key when primary is deleted.
     *
     *       importedKeyCascade - delete rows that import a deleted key
     *       importedKeyRestrict - do not allow delete of primary
     *               key if it has been imported
     *       importedKeySetNull - change imported key to NULL if
     *               its primary key has been deleted
     *
     *FK_NAME String => foreign key identifier (may be null)
     *PK_NAME String => primary key identifier (may be null)
     *  </OL>
     *
     * @param catalog a catalog name; "" retrieves those without a catalog
     * @param schema a schema name pattern; "" retrieves those
     * without a schema
     * @param table a table name
     * @return ResultSet each row is a foreign key column description
     */
    ResultSet getCrossReference(
    String primaryCatalog, String primarySchema, String primaryTable,
    String foreignCatalog, String foreignSchema, String foreignTable
    ) throws SQLException;

    /**
     * Get a description of all the standard SQL types supported by
     * this database. They are ordered by DATA_TYPE and then by how
     * closely the data type maps to the corresponding JDBC SQL type.
     *
     * Each type description has the following columns:
     *  <OL>
     *TYPE_NAME String => Type name
     *DATA_TYPE short => SQL data type from java.sql.Types
     *PRECISION int => maximum precision
     *LITERAL_PREFIX String => prefix used to quote a literal
     *      (may be null)
     *LITERAL_SUFFIX String => suffix used to quote a literal
           (may be null)
     *CREATE_PARAMS String => parameters used in creating
     *      the type (may be null)
     *NULLABLE short => can you use NULL for this type?
     *
     *       typeNoNulls - does not allow NULL values
     *       typeNullable - allows NULL values
     *       typeNullableUnknown - nullability unknown
```

```
     *
     *CASE_SENSITIVE boolean=> is it case sensitive?
     *SEARCHABLE short => can you use “WHERE” based on this type:
     *
     *       typePredNone - No support
     *       typePredChar - Only supported with WHERE .. LIKE
     *       typePredBasic - Supported except for WHERE .. LIKE
     *       typeSearchable - Supported for all WHERE ..
     *
     *UNSIGNED_ATTRIBUTE boolean => is it unsigned?
     *FIXED_PREC_SCALE boolean => can it be a money value?
     *AUTO_INCREMENT boolean => can it be used for an
     *      auto-increment value?
     *LOCAL_TYPE_NAME String => localized version of type name
     *      (may be null)
     *MINIMUM_SCALE short => minimum scale supported
     *MAXIMUM_SCALE short => maximum scale supported
     *SQL_DATA_TYPE int => unused
     *SQL_DATETIME_SUB int => unused
     *NUM_PREC_RADIX int => usually 2 or 10
     *  </OL>
     *
     * @return ResultSet each row is a SQL type description
     */
    ResultSet getTypeInfo() throws SQLException;

    /**
     * TYPE NULLABLE - does not allow NULL values.
     */
    int typeNoNulls = 0;

    /**
     * TYPE NULLABLE - allows NULL values.
     */
    int typeNullable = 1;

    /**
     * TYPE NULLABLE - nullability unknown.
     */
    int typeNullableUnknown = 2;

    /**
     * TYPE INFO SEARCHABLE - No support.
     */
    int typePredNone = 0;

    /**
     * TYPE INFO SEARCHABLE - Only supported with WHERE .. LIKE.
     */
    int typePredChar = 1;

    /**
     * TYPE INFO SEARCHABLE -  Supported except for WHERE .. LIKE.
     */
```

```
int typePredBasic = 2;

/**
 * TYPE INFO SEARCHABLE - Supported for all WHERE ...
 */
int typeSearchable  = 3;

/**
 * Get a description of a table's indices and statistics. They are
 * ordered by NON_UNIQUE, TYPE, INDEX_NAME, and ORDINAL_POSITION.
 *
 * Each index column description has the following columns:
 *  <OL>
 *TABLE_CAT String => table catalog (may be null)
 *TABLE_SCHEM String => table schema (may be null)
 *TABLE_NAME String => table name
 *NON_UNIQUE boolean => Can index values be non-unique?
 *      false when TYPE is tableIndexStatistic
 *INDEX_QUALIFIER String => index catalog (may be null);
 *      null when TYPE is tableIndexStatistic
 *INDEX_NAME String => index name; null when TYPE is
 *      tableIndexStatistic
 *TYPE short => index type:
 *
 *       tableIndexStatistic - this identifies table statistics that are
 *           returned in conjuction with a table's index descriptions
 *       tableIndexClustered - this is a clustered index
 *       tableIndexHashed - this is a hashed index
 *       tableIndexOther - this is some other style of index
 *
 *ORDINAL_POSITION short => column sequence number
 *      within index; zero when TYPE is tableIndexStatistic
 *COLUMN_NAME String => column name; null when TYPE is
 *      tableIndexStatistic
 *ASC_OR_DESC String => column sort sequence, "A" => ascending,
 *      "D" => descending, may be null if sort sequence is not supported;
 *      null when TYPE is tableIndexStatistic
 *CARDINALITY int => When TYPE is tableIndexStatisic then
 *      this is the number of rows in the table; otherwise it is the
 *      number of unique values in the index.
 *PAGES int => When TYPE is  tableIndexStatisic then
 *      this is the number of pages used for the table, otherwise it
 *      is the number of pages used for the current index.
 *FILTER_CONDITION String => Filter condition, if any.
 *      (may be null)
 *  </OL>
 *
 * @param catalog a catalog name; "" retrieves those without a catalog
 * @param schema a schema name pattern; "" retrieves those without a schema
 * @param table a table name
 * @param unique when true, return only indices for unique values;
 *     when false, return indices regardless of whether unique or not
 * @param approximate when true, result is allowed to reflect approximate
 *     or out of data values; when false, results are requested to be
```

```
         *     accurate
         * @return ResultSet each row is an index column description
         */
        ResultSet getIndexInfo(String catalog, String schema, String table,
            boolean unique, boolean approximate)
                    throws SQLException;

        /**
         * INDEX INFO TYPE - this identifies table statistics that are
         * returned in conjuction with a table's index descriptions
         */
        short tableIndexStatistic = 0;

        /**
         * INDEX INFO TYPE - this identifies a clustered index
         */
        short tableIndexClustered = 1;

        /**
         * INDEX INFO TYPE - this identifies a hashed index
         */
        short tableIndexHashed    = 2;

        /**
         * INDEX INFO TYPE - this identifies some other form of index
         */
        short tableIndexOther     = 3;
    }
```

```
// Copyright (c) 1996 Sun Microsystems, Inc. All Rights Reserved.

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

package java.sql;

public interface ResultSetMetaData {

    /**
     * What's the number of columns in the ResultSet?
     *
     * @return the number
     */
    int getColumnCount() throws SQLException;

    /**
     * Is the column automatically numbered, thus read-only?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return true if so
     */
    boolean isAutoIncrement(int column) throws SQLException;

    /**
     * Does a column's case matter?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return true if so
     */
    boolean isCaseSensitive(int column) throws SQLException;

    /**
     * Can the column be used in a where clause?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return true if so
     */
    boolean isSearchable(int column) throws SQLException;

    /**
     * Is the column a cash value?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return true if so
     */
    boolean isCurrency(int column) throws SQLException;

    /**
     * Can you put a NULL in this column?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return columnNoNulls, columnNullable or columnNullableUnknown
     */
```

```
int isNullable(int column) throws SQLException;

/**
 * Does not allow NULL values.
 */
int columnNoNulls = 0;

/**
 * Allows NULL values.
 */
int columnNullable = 1;

/**
 * Nullability unknown.
 */
int columnNullableUnknown = 2;

/**
 * Is the column a signed number?
 *
 * @param column the first column is 1, the second is 2, ...
 * @return true if so
 */
boolean isSigned(int column) throws SQLException;

/**
 * What's the column's normal max width in chars?
 *
 * @param column the first column is 1, the second is 2, ...
 * @return max width
 */
int getColumnDisplaySize(int column) throws SQLException;

/**
 * What's the suggested column title for use in printouts and
 * displays?
 *
 * @param column the first column is 1, the second is 2, ...
 * @return true if so
 */
String getColumnLabel(int column) throws SQLException;

/**
 * What's a column's name?
 *
 * @param column the first column is 1, the second is 2, ...
 * @return column name
 */
String getColumnName(int column) throws SQLException;

/**
 * What's a column's table's schema?
 *
 * @param column the first column is 1, the second is 2, ...
```

```
     * @return schema name or "" if not applicable
     */
    String getSchemaName(int column) throws SQLException;

    /**
     * What's a column's number of decimal digits?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return precision
     */
    int getPrecision(int column) throws SQLException;

    /**
     * What's a column's number of digits to right of decimal?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return scale
     */
    int getScale(int column) throws SQLException;

    /**
     * What's a column's table name?
     *
     * @return table name or "" if not applicable
     */
    String getTableName(int column) throws SQLException;

    /**
     * What's a column's table's catalog name?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return column name or "" if not applicable.
     */
    String getCatalogName(int column) throws SQLException;

    /**
     * What's a column's SQL type?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return SQL type
     */
    int getColumnType(int column) throws SQLException;

    /**
     * What's a column's data source specific type name?
     *
     * @param column the first column is 1, the second is 2, ...
     * @return type name
     */
    String getColumnTypeName(int column) throws SQLException;

    /**
     * Is a column definitely not writable?
     *
```

```
         * @param column the first column is 1, the second is 2, ...
         * @return true if so
         */
        boolean isReadOnly(int column) throws SQLException;

        /**
         * Is it possible for a write on the column to succeed?
         *
         * @param column the first column is 1, the second is 2, ...
         * @return true if so
         */
        boolean isWritable(int column) throws SQLException;

        /**
         * Will a write on the column definitely succeed?
         *
         * @param column the first column is 1, the second is 2, ...
         * @return true if so
         */
        boolean isDefinitelyWritable(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 getXXX and setXXX methods that is described in Sections 7.2 and 7.1. After comparing various example programs we decided that the getXXX/setXXX 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 decided to use the getXXX/setXXX 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.executeUpdate(“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.executeUpdate("{CALL testProcedure(?, ?)}");
      byte x = ih.value;
      String s = sh.value;
}
```

A.1.2 Getting row results using Holder objects

Before executing a statement we can 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.

A.4 Columns numbers in ResultSet.get methods

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 because we were unsatisfied with the readability 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.

A.5 Method overloading for set methods

In an earlier version of the design we used method overloading so that rather than having methods with different names such as setByte, setBoolean, 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.

A.6 That wretched registerOutParameter method

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

A.7 Support for large OUT parameters.

We don't currently support very large OUT parameters. 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 executes. 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.

A.8 Support for GetObject versus getXXX

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, java.sql.Types.INTEGER)).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

A.9 isNull versus wasNull

We had some difficulty in determining a good way of handling SQL NULLs. However by JDBC 0.50 we had designed a ResultSet.isNull method that seemed fairly pleasant to use. The isNull method could be called on any column to check for NULL before (or after) reading the column.

```
if (!ResultSet(isNull(3)) {
       count += ResultSet.getInt(3);
}
```

Unfortunately, harsh reality intervened and it emerged that "isNull" could not be implemented reliably on all databases. Some databases have no separate means for determining if a column is null other than reading the column and they would only permit a given column to be read once. We looked at reading the column value and "remembering" it for later use, but this caused problems when data conversions were required.

After examining a number of different solutions we reluctantly decided to replace the isNull method with the wasNull method. The wasNull method merely reports whether the last value read from the given ResultSet (or CallableStatement) was SQL NULL.

A.10 Use of Java type names v SQL type names.

In an earlier version of the spec we used getXXX and setXXX method for retrieving results and accessing parameters, where the XXX was a SQL type name. In revision 0.70 of JDBC we changed to use getXXX and setXXX methods where the XXX was a Java type name.

Thus for example, getChar was replaced by getString and setSmallInt by setShort.

The new methods have essentially the same semantics as the methods that they replace. However the use of Java type names makes the meaning of each of the methods clearer to Java programmers.

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 = DriverManager.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.getInt(1);
                Numeric b = rs.getNumeric(2);
                char c[] = rs.getString(3).tocharArray();
                boolean d = rs.getBoolean(4);
                String key = rs.getString(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 = DriverManager.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 to 34.
            stmt.setInt(1, 34);
            stmt.setString(2, “count”);
            stmt.executeUpdate();
            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.setString(2, “mirror”);
            stmt.executeUpdate();
            System.out.println(“Updated \”mirror\” row OK.”);

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

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

Appendix C: Implementation notes

C.1 ResultSet lookups by method name

Here is some specimen code that implements the ResultSet.findColumn and (for example) ResultSet.getString using ResultSetMetaData.

```
private java.util.Hashtable s2c;     // Maps strings to column indexes
private ResultSetMetaData md;        // Our metadata object.

public synchronized int findColumn(String columnName)
                                              throws SQLException {
       // Make a mapping cache if we don't already have one.
       if (md == null) {
           md = getMetaData();
           s2c = new java.util.Hashtable();
       }
       // Look for the mapping in our cache.
       Integer x = (Integer)s2c.get(columnName);
       if (x != null) {
           return (x.intValue());
       }
       // OK, we'll have to use metadata.
       for (int i = 1; i < md.getColumnCount(); i++) {
           if (md.getColumnName(i).equalsIgnoreCase(columnName)) {
             // Success!  Add an entry to the cache.
             s2c.put(columnName, new Integer(i));
             return (i);
           }
       }
       throw new SQLException("Column name not found", "S0022");
}

// now the individual get-by-column-name methods are easy:

public String getString(String columnName) throws SQLException {
       return (getString(findColumn(columnName)));
}
```

C.2 Object finalization

Applets are advised to call “close” on the various JDBC objects such as Statement, ResultSet, Connection, when they are done with them. However some applets will forget and some applets may get killed before they can close these objects.

If JDBC drivers have state associated with JDBC objects that needs to get explicitly cleared up, then they should take care to provide “finalize” methods. The garbage collector will call these finalize methods when the objects are found to be garbage and this will give the driver a chance to close (or otherwise clean up) the objects. Note however that there is no guarantee that the garbage collector will ever run, so you can’t rely on the finalizers being called.

Appendix D: Change History

Changes between 0.50 and 0.60:

- Revised the description of transactions to remove references to the obsolete beginTransaction method (9.3).
- Fixed various minor spelling errors, miscapitalizations, and inconsistencies in method names, particularly in the DatabaseMetaData interface.
- Added methods to SQLWarning for managing chains.
- Specified that metadata methods should throw exceptions rather than returning null ResultSets if you try to retrieve catalog information that isn't available. (See comment at the head of DatabaseMetaData.java).
- Moved various historical notes from the main text to Appendix A.
- Added Statement.executeUpdate and modified Statement.execute. Broadly speaking, the execute method is intended for dynamic programming whereas executeQuery and executeUpdate are intended for simple statement execution.
- Rewrote Section 5.3 to clarify security requirements for JDBC driver implementors.
- Added DriverManager.println for logging to the JDBC log stream.
- Removed DriverManager.loadDriver and modified Section 6.6 to advise that you use Class.forName to load drivers (this reflects Java classloader implementation issues.)
- Clarifications in Section 11 but no interface changes.
- Removed ParameterMetadata. The contents were a subset of the information returned by DatabaseMetaData.getProcedureColumns and the latter interface is much more complete.
- Renamed Environment to DriverManager. This name better reflects the final functionality of this class.
- Stated that JDBC will do space padding for CHAR(n) fields (Section 8.3).
- Added section 7.4 to describe data truncation rules.
- Removed ResultSet.getRowNumber. This is easy for application to do themselves.
- Changed constant values for transaction isolation levels to be the same as ODBC.
- Removed ResultSet.getRowCount. Several reviewers had expressed concern that this was prohibitively expensive to implement.
- Removed ResultSet.getColumnCount. This method is unnecessary as there is already a ResultSetMetaData.getColumnCount method and programmers requiring to know the number of columns are also likely to require other column metadata.
- By popular demand added additional ResultSet methods to retrieve values by column name.
- Made ResultSet an abstract class (rather than an interface) so we could provide default implementations of the methods that take column names. (Note: we are nervous about the implications of this and are still reviewing this decision.)
- Added Statement.setCursorName
- Added Java "doc comment" stylized comments to many of the API descriptions.

Changes between 0.60 and 0.70

- Added separate ResultSet warning chain (and access methods) in addition to the Statement warning chain.
- Clarified the behaviour of getAscciStream and getUnicodeStream on ResultSets.
- Clarified the behaviour of setAsciiStream and setUnicodeStream on PreparedStatements.
- Stated explicitly that LONGVARCHAR and LONGVARBINARY are not supported for stored procedure OUT parameters (see 7.3.3 and 14.2).
- Removed getLongVarChar and getLongVarBinary from CallableStatement.
- Changed ResultSet back to an interface (rather than an abstract class). We now provide an specimen implementation of ResultSet.findColumn as reference material in Appendix C.
- Replaced ResultSet.isNull with ResultSet.wasNull. We prefer the isNull style, but early JDBC implementors discovered that it was impossible to implement isNull reliably on some databases for some data types. Unfortunately on these databases it is necessary to first attempt a read and then to check if the read returned null.
- Similarly replaced CallableStatement.isNull with CallableStatement.wasNull.
- Added Driver.jdbcCompliant to report if driver is fully JDBC COMPLIANT (tm).
- Added specification of jdbc:odbc sub-protocol URL syntax (6.3.6).
- Added LIKE escape characters (11.4).
- Added Driver.acceptURL and DriverManager.getDriver
- Stated that auto-close only applied to PreparedStatements, CallableStatements, and ResultSets and doesn't affect simple Statements (9.3).
- Added Driver.getPropertyInfo method and DriverPropertyInfo class to allow generic GUI applications to discover what connection properties a driver expects.
- Removed the SQLWarning type NullData. The wasNull method can be used to check if null data has been read.
- Clarified that ResultSet warnings are cleared on each call to "next".
- Clarified that Statement warnings are cleared each time a statement is (re)executed.
- Replaced all methods of the format getXXX or setXXX with analogous methods whose names are based on the Java type returned rather than the SQL type. Thus, for example, getTinyInt becomes getByte. See Section 7 and also Appendix A.10. This is a fairly substantial change which we made because of reviewer input that showed widespread confusion over the use of SQL types in the method names, and a widely stated preference for Java type names.
- Added additional getObject/setObject methods and extra explanatory text (14.2).
- Added sections to clarify when data conversions are and are not performed (7.1.1, 7.2.1, and 7.3.1).
- Added extra tables to clarify Java to SQL type mappings (Tables 3 and 5).

Changes between 0.70 and 0.95

- Changed the default mapping for SQL FLOAT to be Java double or Double (Tables 1, 2, 4).
- Changed the default mapping for Java float and Float to be SQL REAL (Tables 3, 5).
- Clarified that network protocol security is not a JDBC issue (Section 5).
- Renamed SQL Escape section, explaining extensions and clarifying relationship to SQL levels defined by ANSI and Microsoft's ODBC.
- Added sentence clarifying positioned updates (Section 10).
- Removed old Section 7.1 "Parameters and results in ODBC" and various other comparisons with ODBC. While this was useful for early reviewers, it's not terribly useful in the finished spec.
- Clarified that all JDBC compliant drivers must support CallableStatement but need not support the methods relating to OUT parameters (3.2, 12.1).
- Replaced disableAutoClose with setAutoClose
- Added Connection.getAutoClose and Connection.getAutoCommit
- Added Table 6 to document conversions in PreparedStatement.setObject.
- Removed the versions of ResultSet.getObject and CallableStatement.getObject that took a target SQL type. (We still retain the simpler getObject methods.) These methods were hard to implement and hard to explain and there appeared to be no real demand for them.
- Added advice on finalizers in Appendix C.2.

Changes between 0.95 and 1.00

- Clarified that when retrieving ResultSet column values by column name then the first matching column will be returned in cases where there are several columns with the same name.
- Clarified that if you pass a Java null to one of the PreparedSattement.setXXX methods then this will cause a SQL NULL to be sent to the database.
- Added an extra column SQL_DATA_TYPE to the DatabaseMetaData.getColumns results.
- Removed the transaction isolation level TRANSCTION_VERSIONING
- Noted that if a stored procedure returns both ResultSets and OUT parameters then the ResultSets must be read and processed before the OUT parameters are read (7.3).
- Noted that for maximum portability the columns of a ResultSet should be retrieved in left-to-right order (7.1).
- Reworded some of the introduction that was written before we first released the spec.