JDBC™ Guide: Getting Started

Table of Contents

1 Introduction . . . 1

1.1 What Is JDBC™? . . . 1
- 1.1.1 What Does JDBC Do? . . . 2
- 1.1.2 JDBC Is a Low-level API and a Base for Higher-level APIs . . . 3
- 1.1.3 JDBC versus ODBC and other APIs . . . 4
- 1.1.4 Two-tier and Three-tier Models . . . 5
- 1.1.5 SQL Conformance . . . 6

1.2 JDBC Products . . . 7
- 1.2.1 JavaSoft Framework . . . 8
- 1.2.2 JDBC Driver Types . . . 9
- 1.2.3 Obtaining JDBC Drivers . . . 10
- 1.2.4 Other Products . . . 10

2 Connection . . . 13

2.1 Overview . . . 13
- 2.1.1 Opening a Connection . . . 13
- 2.1.2 URLs in General Use . . . 14
- 2.1.3 JDBC URLs . . . 15
- 2.1.4 The "odbc" Subprotocol . . . 17
- 2.1.5 Registering Subprotocols . . . 17
- 2.1.6 Sending SQL Statements . . . 18
- 2.1.7 Transactions . . . 19
- 2.1.8 Transaction Isolation Levels . . . 20

3 DriverManager . . . 21

3.1 Overview . . . 21
- 3.1.1 Keeping Track of Available Drivers . . . 21
- 3.1.2 Establishing a Connection . . . 23

4 Statement . . . 25

4.1 Overview . . . 25
- 4.1.1 Creating `Statement` Objects . . . 25
- 4.1.2 Executing `Statement` Objects . . . 26
- 4.1.3 Using the Method `Execute` . . . 27
- 4.1.4 `Statement` Completion . . . 29
- 4.1.5 SQL Escape Syntax in `Statement` Objects . . . 30

5 ResultSet 33

5.1 Overview 33
5.1.1 Rows and Cursors 34
5.1.2 Columns 34
5.1.3 Data Types and Conversions 35
5.1.4 Using Streams for Very Large Row Values 37
5.1.5 `NULL` Result Values 38
5.1.6 Optional or Multiple Result Sets 38

6 PreparedStatement 39

6.1 Overview 39
6.1.1 Creating `PreparedStatement` Objects 40
6.1.2 Passing `IN` Parameters 40
6.1.3 Data Type Conformance on IN Parameters 41
6.1.4 Using `setObject` 42
6.1.5 Sending `SQL NULL` as an IN parameter 42
6.1.6 Sending Very Large IN Parameters 42

7 CallableStatement 45

7.1 Overview 45
7.1.1 Creating a `CallableStatement` Object 46
7.1.2 IN and OUT Parameters 46
7.1.3 INOUT Parameters 47
7.1.4 Retrieve OUT Parameters after Results 48
7.1.5 Retrieving NULL Values as OUT Parameters 48

8 Mapping SQL and Java Types 49

8.1 Overview 49
8.2 Mapping SQL Data Types into Java 49
8.3 SQL Types 50
8.3.1 `CHAR`, `VARCHAR`, and `LONGVARCHAR` 50
8.3.2 `DECIMAL` and `NUMERIC` 50
8.3.3 `BINARY`, `VARBINARY`, and `LONGVARBINARY` 51
8.3.4 `BIT` 51
8.3.5 `TINYINT`, `SMALLINT`, `INTEGER`, and `BIGINT` 51
8.3.6 `REAL`, `FLOAT`, and `DOUBLE` 52
8.3.7 `DATE`, `TIME`, and `TIMESTAMP` 52
8.4 Examples of Mapping 53
8.4.1 Simple SQL Statement 53
8.4.2 SQL Statement with IN Parameters 54
8.4.3 SQL Statement with INOUT Parameters 54
8.5 Tables for Data Type Mapping 56
8.5.1 SQL Types Mapped to Java Types 57
8.5.2 Java Types Mapped to SQL Types 58
8.5.3 SQL Types Mapped to Java Object Types 59
8.5.4 Java Object Types Mapped to SQL Types 60

8.5.5 Conversions by `setObject` 61
8.5.6 SQL Types Retrieved by `ResultSet.getXXX` Methods 62

9 Sample Code 63

10 JDBC-ODBC Release Notes. 69

10.1 JDBC-ODBC Bridge 69
10.1.1 What Is the JDBC-ODBC Bridge? 69
10.1.2 What Version of ODBC Is Supported? 69
10.1.3 The Bridge Implementation 69
10.1.4 Installation 70
10.2 Using the Bridge. 70
10.2.1 Using the Bridge from an Applet. 70
10.2.2 Most Browsers Do Not Support the Bridge 70
10.2.3 Tested Configurations. 71
10.2.4 ODBC Drivers Known to Work with the Bridge 71
10.2.5 ODBC Driver Incompatibilities 71
10.2.6 What Is the JDBC URL Supported by the Bridge? 71
10.2.7 Debugging 72
10.3 General Notes 72

1 Introduction

THIS introduction is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

1.1 What Is JDBC™?

JDBC™ is a Java™ API for executing SQL statements. (As a point of interest, JDBC is a trademarked name and is not an acronym; nevertheless, JDBC is often thought of as standing for "Java Database Connectivity".) It consists of a set of classes and interfaces written in the Java programming language. JDBC provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.

Using JDBC, it is easy to send SQL statements to virtually any relational database. In other words, with the JDBC API, it isn't necessary to write one program to access a Sybase database, another program to access an Oracle database, another program to access an Informix database, and so on. One can write a single program using the JDBC API, and the program will be able to send SQL statements to the appropriate database. And, with an application written in the Java programming language, one also doesn't have to worry about writing different applications to run on different platforms. The combination of Java and JDBC lets a programmer write it once and run it anywhere.

Java, being robust, secure, easy to use, easy to understand, and automatically downloadable on a network, is an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different databases. JDBC is the mechanism for doing this.

JDBC extends what can be done in Java. For example, with Java and the JDBC API, it is possible to publish a web page containing an applet that uses information obtained from a remote database. Or an enterprise can use JDBC to connect all its employees (even if they are using a conglomeration of Windows, Macintosh, and UNIX machines) to one or more internal databases via an intranet. With more and more programmers using the Java programming language, the need for easy database access from Java is continuing to grow.

MIS managers like the combination of Java and JDBC because it makes disseminating information easy and economical. Businesses can continue to use their installed databases and access information easily even if it is stored on different database management systems. Development time for new applications is short. Installation and version control are greatly simplified. A programmer can write an application or an update once, put it on the server, and everybody has access to the latest version. And for businesses selling information services, Java and JDBC offer a better way of getting out information updates to external customers.

1.1.1 What Does JDBC Do?

Simply put, JDBC makes it possible to do three things:

1. establish a connection with a database
2. send SQL statements
3. process the results.

The following code fragment gives a basic example of these three steps:

```
Connection con = DriverManager.getConnection (
                     "jdbc:odbc:wombat", "login", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
    int x = getInt("a");
    String s = getString("b");
    float f = getFloat("c");
}
```

1.1.2 JDBC Is a Low-level API and a Base for Higher-level APIs

JDBC is a “low-level” interface, which means that it is used to invoke (or “call”) SQL commands directly. It works very well in this capacity and is easier to use than other database connectivity APIs, but it was designed also to be a base upon which to build higher-level interfaces and tools. A higher-level interface is “user-friendly,” using a more understandable or more convenient API that is translated behind the scenes into a low-level interface such as JDBC. At the time of this writing, two kinds of higher-level APIs are under development on top of JDBC:

1. an embedded SQL for Java. At least one vendor plans to build this. DBMSs implement SQL, a language designed specifically for use with databases. JDBC requires that the SQL statements be passed as Strings to Java methods. An embedded SQL preprocessor allows a programmer to instead mix SQL statements directly with Java: for example, a Java variable can be used in a SQL statement to receive or provide SQL values. The embedded SQL preprocessor then translates this Java/SQL mix into Java with JDBC calls.
2. a direct mapping of relational database tables to Java classes. JavaSoft and others have announced plans to implement this. In this “object/relational” mapping, each row of the table becomes an instance of that class, and each column value corresponds to an attribute of that instance. Programmers can then operate directly on Java objects; the required SQL calls to fetch and store data are automatically generated “beneath the covers.” More sophisticated mappings are also provided, for example, where rows of multiple tables are combined in a Java class.

As interest in JDBC has grown, more developers have been working on JDBC-based tools to make building programs easier, as well. Programmers have also been writing applications that make accessing a database easier for the end user. For example, an application might present a menu of database tasks from which to choose. After a task is selected, the application presents prompts and blanks for filling in information needed to carry out the selected task. With the requested input typed in, the application then automatically invokes the necessary SQL commands. With the help of such an application, users can perform database tasks even when they have little or no knowledge of SQL syntax.

1.1.3 JDBC versus ODBC and other APIs

At this point, Microsoft's ODBC (Open DataBase Connectivity) API is probably the most widely used programming interface for accessing relational databases. It offers the ability to connect to almost all databases on almost all platforms. So why not just use ODBC from Java?

The answer is that you *can* use ODBC from Java, but this is best done with the help of JDBC in the form of the JDBC-ODBC Bridge, which we will cover shortly. The question now becomes, "Why do you need JDBC?" There are several answers to this question:

1. ODBC is not appropriate for direct use from Java because it uses 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.
2. A literal translation of the ODBC C API into a Java API would not be desirable. For example, Java has no pointers, and ODBC makes copious use of them, including the notoriously error-prone generic pointer "void *". You can think of JDBC as ODBC translated into an object-oriented interface that is natural for Java programmers.
3. ODBC is hard to learn. It mixes simple and advanced features together, and it has complex options even for simple queries. JDBC, on the other hand, was designed to keep simple things simple while allowing more advanced capabilities where required.
4. A Java API like JDBC is needed in order to enable a "pure Java" solution. When ODBC is used, the ODBC driver manager and drivers must be manually installed on every client machine. When the JDBC driver is written completely in Java, however, JDBC code is automatically installable, portable, and secure on all Java platforms from network computers to mainframes.

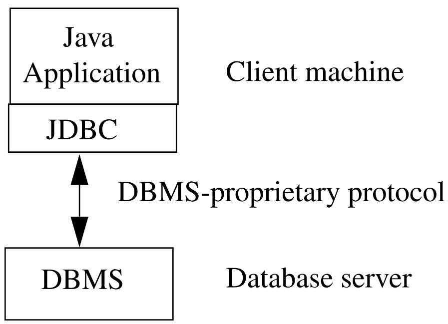
In summary, the JDBC API is a natural Java interface to the basic SQL abstractions and concepts. It builds on ODBC rather than starting from scratch, so programmers familiar with ODBC will find it very easy to learn JDBC. JDBC retains the basic design features of ODBC; in fact, both interfaces are based on the X/Open SQL CLI (Call Level Interface). The big difference is that JDBC builds on and reinforces the style and virtues of Java, and, of course, it is easy to use.

More recently, Microsoft has introduced new APIs beyond ODBC: RDO, ADO, and OLE DB. These designs move in the same direction as JDBC in many ways, that is, in being an object-oriented database interface based on classes that can be implemented on ODBC. However, we did not see compelling functionality in any of these interfaces to make them an alternative basis to ODBC, especially with the ODBC driver market well-established. Mostly they represent a thin veneer on ODBC. This is not to say that JDBC does not need to evolve from the initial release; however, we feel that most new functionality belongs in higher-level APIs such as the object/relational mappings and embedded SQL mentioned in the previous section.

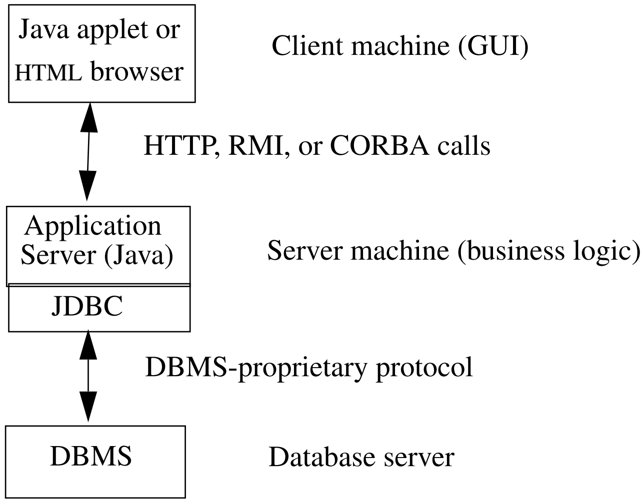
1.1.4 Two-tier and Three-tier Models

The JDBC API supports both two-tier and three-tier models for database access.

In the two-tier model, a Java applet or application talks directly to the database. This requires a JDBC driver that can communicate with the particular database management system being accessed. A user's SQL statements are delivered to the database, and the results of those statements are sent back to the user. The database may be located on another machine to which the user is connected via a network. This is referred to as a *client/server* configuration, with the user's machine as the client, and the machine housing the database as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.



In the three-tier model, commands are sent to a "middle tier" of services, which then send SQL statements to the database. The database processes the SQL statements and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that when there is a middle tier, the user can employ an easy-to-use higher-level API which is translated by the middle tier into the appropriate low-level calls. Finally, in many cases the three-tier architecture can provide performance advantages.



Until now the middle tier has typically been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code, it is becoming practical to implement the middle tier in Java. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features. JDBC is important to allow database access from a Java middle tier.

1.1.5 SQL Conformance

Structured Query Language (SQL) is the standard language for accessing relational databases. One area of difficulty is that although most DBMSs (DataBase Management Systems) use a standard form of SQL for basic functionality, they do not conform to the more recently-defined standard SQL syntax or semantics for more advanced functionality. For example, not all databases support stored procedures or outer joins, and those that do are not consistent with each other. It is hoped that the portion of SQL that is truly standard will expand to include more and more functionality. In the meantime, however, the JDBC API must support SQL as it is.

One way the JDBC API deals with this problem is to allow any query string to be passed through to an underlying DBMS driver. This means that an application is free to use as much SQL functionality as desired, but it runs 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 designed for specific DBMSs (for document or image queries, for example).

A second way JDBC deals with problems of SQL conformance is to provide ODBC-style escape clauses, which are discussed in section 4.1.5, "SQL Escape Syntax in Statement Objects."

. The escape syntax provides a standard JDBC syntax for several of the more common areas of SQL divergence. For example, there are escapes for date literals and for stored procedure calls.

For complex applications, JDBC deals with SQL conformance in a third way. It provides descriptive information about the DBMS by means of the `DatabaseMetaData` interface so that applications can adapt to the requirements and capabilities of each DBMS.

Because the JDBC API will be used as a base API for developing higher-level database access tools and APIs, it also has to address the problem of conformance for anything built on it. The designation "JDBC COMPLIANT™" was created to set a standard level of JDBC functionality on which users can rely. In order to use this designation, a driver must support at least ANSI SQL-2 Entry Level. (ANSI SQL-2 refers to the standards adopted by the American National Standards Institute in 1992. Entry Level refers to a specific list of SQL capabilities.) Driver developers can ascertain that their drivers meet these standards by using the test suite available with the JDBC API.

The "JDBC COMPLIANT™" designation indicates that a vendor's JDBC implementation has passed the conformance tests provided by JavaSoft. These conformance tests check for the existence of all of the classes and methods defined in the JDBC API, and check as much as possible that the SQL Entry Level functionality is available. Such tests are not exhaustive, of course, and JavaSoft is not currently branding vendor implementations, but this compliance definition provides some degree of confidence in a JDBC implementation. With wider and wider acceptance of the JDBC API by database vendors, connectivity vendors, Internet service vendors, and application writers, JDBC is quickly becoming the standard for Java database access.

1.2 JDBC Products

The JDBC API is a natural choice for Java developers because it offers easy database access for Java applications and applets.

At the time of this writing, a number of JDBC-based products have already been deployed or are under development. Some description of these products will put JDBC in perspective. Of course, the information in this section will quickly become dated, so the reader should consult the JDBC web page for the latest information. It can be found by navigating from the following URL:

```
http://www.javasoft.com/products/jdbc
```

1.2.1 JavaSoft Framework

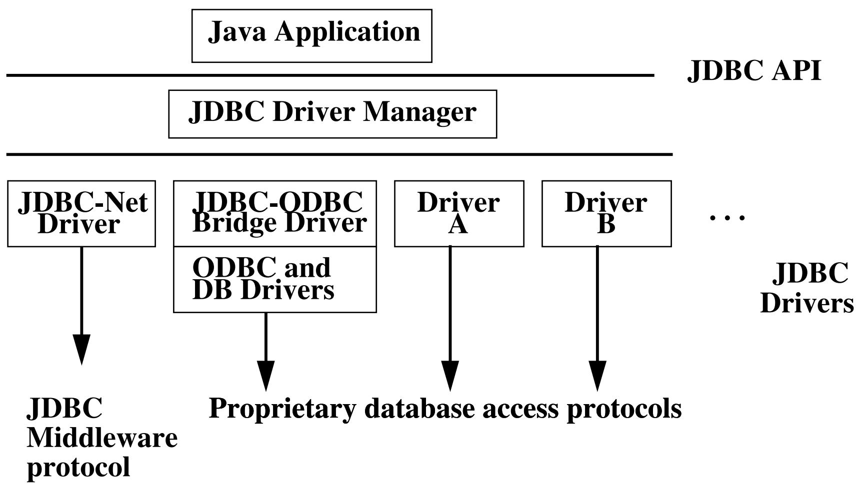
JavaSoft provides three JDBC product components as part of the Java Developer's Kit (JDK):

- the JDBC driver manager,
- the JDBC driver test suite, and
- the JDBC-ODBC bridge.

The JDBC driver manager is the backbone of the JDBC architecture. It actually is quite small and simple; its primary function is to connect Java applications to the correct JDBC driver and then get out of the way.

The JDBC driver test suite provides some confidence that JDBC drivers will run your program. Only drivers that pass the JDBC driver test suite can be designated JDBC COMPLIANT™.

The JDBC-ODBC bridge allows ODBC drivers to be used as JDBC drivers. It was implemented as a way to get JDBC off the ground quickly, and long term will provide a way to access some of the less popular DBMSs if JDBC drivers are not implemented for them.



1.2.2 JDBC Driver Types

The JDBC drivers that we are aware of at this time fit into one of four categories:

1. *JDBC-ODBC bridge plus ODBC driver:* The JavaSoft bridge product provides JDBC access via ODBC drivers. Note that ODBC binary code, and in many cases database client code, must be loaded on each client machine that uses this driver. As a result, this kind of driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.
2. *Native-API partly-Java driver:* This kind of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.
3. *JDBC-Net pure Java driver:* This driver translates JDBC calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases. The specific protocol used depends on the vendor. In general, this is the most flexible JDBC alternative. It is likely that all vendors of this solution will provide products suitable for Intranet use. In order for these products to also support Internet access, they must handle the additional requirements for security, access through firewalls, and so on, that the Web imposes. Several vendors are adding JDBC drivers to their existing database middleware products.
4. *Native-protocol pure Java driver:* This kind of driver converts JDBC calls into the network protcol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary, the database vendors themselves will be the primary source, and several database vendors have these in progress.

Eventually, we expect that driver categories 3 and 4 will be the preferred way to access databases from JDBC. Driver categories 1 and 2 are interim solutions where direct pure Java drivers are not yet available. Category 4 is in some sense the ideal, although there are a few cases where Category 3 may be better. (Category 3 might be preferred if a thin DBMS-independent client is desired or if a DBMS-independent protocol is standardized and implemented directly by many DBMS vendors.)

The following chart shows the four categories and their properties:

DRIVER CATEGORY	ALL JAVA?	NET PROTOCOL
1 - JDBC-OCBC Bridge	No	Direct
2 - Native API as basis	No	Direct
3 - JDBC-Net	Yes	Requires Connector
4 - Native protocol as basis	Yes	Direct

1.2.3 Obtaining JDBC Drivers

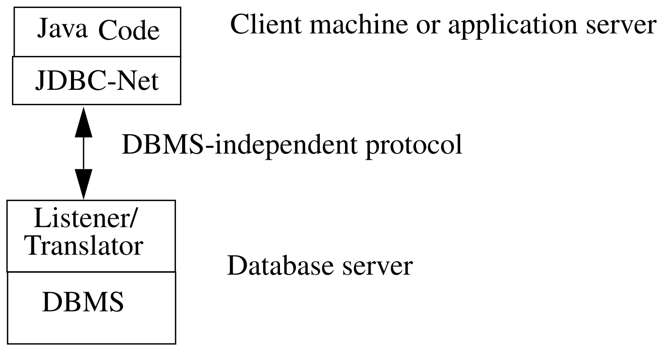
At the time of this writing, there are dozens of drivers in Category 1: ODBC drivers that can be used with JavaSoft's bridge. There are currently about a dozen Category 2 drivers built on top of native APIs for DBMSs. There are a few Category 3 drivers. Currently there are at least two Category 4 drivers, but by the end of 1997, we expect that there will be Category 4 drivers for all of the major DBMSs.

To get the latest information on drivers, check the JDBC web page at `http://www.javasoft.com/products/jdbc`. The first vendors with Category 3 drivers available were SCO, Open Horizon, Visigenic, and WebLogic. JavaSoft and Intersolv, a leading database connectivity vendor, worked together to produce the JDBC-ODBC Bridge and the JDBC Driver Test Suite.

1.2.4 Other Products

Various JDBC application development tools are under way. Watch the JavaSoft pages for updates.

JavaSoft or a standards group may attempt to standardize on a network protocol that is DBMS-independent. In that case, JavaSoft could bundle the "client side” implementation of the protocol with the JDK (Java Developer’s Kit), and various vendors could provide the server side:.



2 Connection

This overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

2.1 Overview

A `Connection` object represents a connection with a database. A connection session includes the SQL statements that are executed and the results that are returned over that connection. A single application can have one or more connections with a single database, or it can have connections with many different databases.

2.1.1 Opening a Connection

The standard way to establish a connection with a database is to call the method `DriverManager.getConnection`. This method takes a string containing a URL. The `DriverManager` class, referred to as the JDBC management layer, attempts to locate a driver than can connect to the database represented by that URL. The `DriverManager` class maintains a list of registered `Driver` classes, and when the method `getConnection` is called, it checks with each driver in the list until it finds one that can connect to the database specified in the URL. The `Driver` method `connect` uses this URL to actually establish the connection.

A user can bypass the JDBC management layer and call `Driver` methods directly. This could be useful in the rare case that two drivers can connect to a database and the user wants to explicitly select a particular driver. Normally, however, it is much easier to just let the `DriverManager` class handle opening a connection.

The following code exemplifies opening a connection to a database located at the URL `"jdbc:odbc:wombat"` with a user ID of `"oboy"` and `"12Java"` as the password :

```
String url = "jdbc:odbc:wombat";
Connection con = DriverManager.getConnection(url, "oboy", "12Java");
```

2.1.2 URLs in General Use

Since URLs often cause some confusion, we will first give a brief explanation of URLs in general and then go on to a discussion of JDBC URLs.

A URL (Uniform Resource Locator) gives information for locating a resource on the Internet. It can be thought of as an address. For general use, a URL is made up of three parts, with only the first part being required for all URLs. (Boldface in the examples below is used to indicate the part being described; it is not part of the URL.)

A URL has three parts:

1. **Protocol used to access the information.** The protocol is always followed by a colon. Some common protocols are `ftp`, which specifies "file transfer protocol," and `http`, which specifies "hypertext transfer protocol." If the protocol is `file`, it indicates that the resource is in a local file system rather than on the Internet.

   ```
   ftp://javasoft.com/docs/JDK-1_apidocs.zip
   http://java.sun.com/products/JDK/1.1
   file:/home/haroldw/docs/tutorial.html
   ```

2. **Host information.** This part gives the information needed to find and access the host where the resource resides. The host information begins with a double slash ("//") if this is an Internet application, such as ftp or http, and a single slash ("/") if it is not. The host information ends with a single slash ("/"). Host information is itself divided into three parts:

   - Domain name of the host if the resource resides on the Internet; if the resource is a local file, there is no host name. Instead there is just the path of the file.
   - User login name and password, which are included if needed.
   - Port number, which is included if needed. A port number follows the host name and a colon (":").

The most common case is to have double slashes and only the hostname:

```
http://java.sun.com
```

The following URL contains the port number 80:

```
http://java.sun.com:80/doc/tutorial.html
```

The following is an example of a URL with a login name `"happy"` and password `"1234"` included as part of the hostname:

```
http://netsmile.grin.com."happy"."1234"/news/latest
```

In the domain name `java.sun.com`, `com` indicates that `java.sun` is a commercial venture. Some other designations are `edu` for an educational institution, `org` for a non-profit organization, and `gov` for governmental organization.

3. **Path of what is to be accessed.** In the following example, `products` and `JDK` are directories, and `1.0.2` is a file. This URL gives the location of the Java Developer's Kit, version 1.0.2:

```
http://java.sun.com/products/JDK/1.0.2
```

2.1.3 JDBC URLs

A JDBC URL provides a way of identifying a database so that the appropriate driver will recognize it and establish a connection with it. A driver needs to understand only one URL naming syntax and can happily reject any other URLs that are presented to it. It is the driver writers themselves who determine the format of a JDBC URL. The first part will always be `jdbc`. The second part will be the subprotocol, which the driver writer provides. The rest of a JDBC URL is the datasource. Information needed to access the data source, such as the user's login name and password, may be part of the JDBC URL, or it may be supplied separately. Users trying to connect to a database just follow the format provided with a driver and supply the information needed to access a database. JDBC's role is simply to recommend some conventions for driver writers to use in structuring JDBC URLs.

Since JDBC URLs are used with various kinds of drivers, the conventions are of necessity very flexible. First, they allow different drivers to use different schemes for naming databases. The `odbc` subprotocol, for example, lets the URL contain attribute values after the subname (but does not require them).

Second, JDBC URLs allow driver writers to encode all necessary connection information within them. This makes it possible, for example, for an applet that wants to talk to a given database to open the database connection without requiring the user to do any system administration chores.

Third, JDBC URLs allow a level of indirection. This means that the JDBC URL may refer to a logical host or database name that is dynamically translated to the actual name by a network naming system. This allows system administrators to avoid specifying particular hosts as part of the JDBC name. There are a number of different network name services (such as DNS, NIS, and DCE), and there is no restriction about which ones can be used.

Since the standard URL naming mechanism already provides many of the features needed in JDBC URLs, the JDBC URL conventions just add a new syntax. The standard syntax for JDBC URLs is:

```
jdbc:<subprotocol>:<subname>
```

A JDBC URL has three parts, which are separated by colons:

1. `jdbc` is the protocol. The protocol in a JDBC URL is always `jdbc`.

2. `<subprotocol>` is usually the driver or the database connectivity mechanism, which may be supported by one or more drivers. A prominent example of a subprotocol name is `odbc`, which has been reserved for URLs that specify ODBC-style data source names. For example, to access a database through a JDBC-ODBC bridge, one might use a URL such as the following:

   ```
   jdbc:odbc:fred
   ```

   In this example, the subprotocol is `odbc`, and the subname `fred` is a local ODBC data source.

3. `<subname>` is a way to identify the database. The subname can vary, depending on the subprotocol, and it can have a subsubname with any internal syntax the driver writer chooses. The point of a subname is to give enough information to locate the database. In the previous example, `fred` is enough because ODBC provides the remainder of the information. A database on a remote server re-

quires more information, however. If the database is to be accessed over the Internet, for example, the network address should be included in the JDBC URL as part of the subname and should follow the standard URL naming convention of `//hostname:port/subsubname`. Supposing that `dbnet` is a protocol for connecting to a host on the Internet, a JDBC URL might look like this:

```
jdbc:dbnet://wombat:356/fred
```

2.1.4 The "odbc" Subprotocol

The subprotocol `odbc` is a special case. It has been reserved for URLs that specify ODBC-style data source names and has the special feature of allowing any number of attribute values to be specified after the subname (the data source name). The full syntax for the odbc subprotocol is:

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

Thus all of the following are valid `jdbc:odbc` names:

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

2.1.5 Registering Subprotocols

A driver developer can reserve a name to be used as the subprotocol in a JDBC URL. When the `DriverManager` class presents this name to its list of registered drivers, the driver for which this name is reserved should recognize it and establish a connection to the database it identifies. For example, `odbc` is reserved for the JDBC-ODBC Bridge. If there were, for another example, a Miracle Corporation, it might want to register "miracle" as the subprotocol for the JDBC driver that connects to its Miracle DBMS so that no one else would use that name.

JavaSoft is acting as an informal registry for JDBC subprotocol names. To register a subprotocol name, send email to:

```
jdbc@wombat.eng.sun.com
```

2.1.6 Sending SQL Statements

Once a connection is established, it is used to pass SQL statements to its underlying database. JDBC does not put any restrictions on the kinds of SQL statements that can be sent; this provides a great deal of flexibility, allowing the use of database-specific statements or even non-SQL statements. It requires, however, that the user be responsible for making sure that the underlying database can process the SQL statements being sent and suffer the consequences if it cannot. For example, an application that tries to send a stored procedure call to a DBMS that does not support stored procedures will be unsuccessful and generate an exception. JDBC requires that a driver provide at least ANSI SQL-2 Entry Level capabilities in order to be designated JDBC COMPLIANT™. This means that users can count on at least this standard level of functionality.

JDBC provides three classes for sending SQL statements to the database, and three methods in the `Connection` interface create instances of these classes. These classes and the methods which create them are listed below:

1. `Statement`- -created by the method `createStatement`. A `Statement` object is used for sending simple SQL statements.
2. `PreparedStatement`- -created by the method `prepareStatement`. A `PreparedStatement` object is used for SQL statements that take one or more parameters as input arguments (IN parameters). `PreparedStatement` has a group of methods which set the value of IN parameters, which are sent to the database when the statement is executed. Instances of `PreparedStatement` extend `Statement` and therefore include `Statement` methods. A `PreparedStatement` object has the potential to be more efficient than a `Statement` object because it has been pre-compiled and stored for future use.
3. `CallableStatement`- -created by the method `prepareCall`. `CallableStatement` objects are used to execute SQL stored procedures- -a group of SQL statements that is called by name, much like invoking a function. A `CallableStatement` object inherits methods for handling IN parameters from `PreparedStatement`; it adds methods for handling OUT and INOUT parameters.

The following list gives a quick way to determine which `Connection` method is appropriate for creating different types of SQL statements:

`createStatement` method is used for

- simple SQL statements (no parameters)

`prepareStatement` method is used for

- SQL statements with one or more IN parameters
- simple SQL statements that are executed frequently

`prepareCall` method is used for

- call to stored procedures

2.1.7 Transactions

A transaction consists of one or more statements that have been executed, completed, and then either committed or rolled back. When the method `commit` or `rollback` is called, the current transaction ends and another one begins.

A new connection is in auto-commit mode by default, meaning that when a statement is completed, the method `commit` will be called on that statement automatically. In this case, since each statement is committed individually, a transaction consists of only one statement. If auto-commit mode has been disabled, a transaction will not terminate until the method `commit` or `rollback` is called explicitly, so it will include all the statements that have been executed since the last invocation of the `commit` or `rollback` method. In this second case, all the statements in the transaction are committed or rolled back as a group.

The method `commit` makes permanent any changes an SQL statement makes to a database, and it also releases any locks held by the transaction. The method `rollback` will discard those changes.

Sometimes a user doesn't want one change to take effect unless another one does also. This can be accomplished by disabling auto-commit and grouping both updates into one transaction. If both updates are successful, then the `commit` method is called, making the effects of both updates permanent; if one fails or both fail, then the `rollback` method is called, restoring the values that existed before the updates were executed.

Most JDBC drivers will support transactions. In fact, a JDBC-compliant driver must support transactions. `DatabaseMetaData` supplies information describing the level of transaction support a DBMS provides.

2.1.8 Transaction Isolation Levels

If a DBMS supports transaction processing, it will have some way of managing potential conflicts that can arise when two transactions are operating on a database at the same time. A user can specify a transaction isolation level to indicate what level of care the DBMS should exercise in resolving potential conflicts. For example, what happens when one transaction changes a value and a second transaction reads that value before the change has been committed or rolled back? Should that be allowed, given that the changed value read by the second transaction will be invalid if the first transaction is rolled back? A JDBC user can instruct the DBMS to allow a value to be read before it has been committed ("dirty reads") with the following code, where *`con`* is the current connection:

```
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

The higher the transaction isolation level, the more care is taken to avoid conflicts. The `Connection` interface defines five levels, with the lowest specifying that transactions are not supported at all and the highest specifying that while one transaction is operating on a database, no other transactions may make any changes to the data read by that transaction. Typically, the higher the level of isolation, the slower the application executes (due to increased locking overhead and decreased concurrency between users). The developer must balance the need for performance with the need for data consistency when making a decision about what isolation level to use. Of course, the level that can actually be supported depends on the capabilities of the underlying DBMS.

When a new `Connection` object is created, its transaction isolation level depends on the driver, but normally it is the default for the underlying database. A user may call the method `setIsolationLevel` to change the transaction isolation level, and the new level will be in effect for the rest of the connection session. To change the transaction isolation level for just one transaction, one needs to set it before the transaction begins and reset it after the transaction terminates. Changing the transaction isolation level during a transaction is not recommended, for it will trigger an immediate call to the method `commit`, causing any changes up to that point to be made permanent.

3 DriverManager

This overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

3.1 Overview

THE `DriverManager` class is the management layer of JDBC, working between the user and the drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. In addition, the `DriverManager` class attends to things like driver login time limits and the printing of log and tracing messages.

For simple applications, the only method in this class that a general programmer needs to use directly is `DriverManager.getConnection`. As its name implies, this method establishes a connection to a database. JDBC allows the user to call the `DriverManager` methods `getDriver`, `getDrivers`, and `registerDriver` as well as the `Driver` method `connect`, but in most cases it is better to let the `DriverManager` class manage the details of establishing a connection.

3.1.1 Keeping Track of Available Drivers

The `DriverManager` class maintains a list of `Driver` classes that have registered themselves by calling the method `DriverManager.registerDriver`. All `Driver` classes should be written with a static section that creates an instance of the class and then registers it with the `DriverManager` class when it is loaded. Thus, a user would not normally call `DriverManager.registerDriver` directly; it should be called automatically by a driver when it is loaded. A `Driver` class is loaded, and therefore automatically registered with the `DriverManager`, in two ways:

1. By calling the method `Class.forName`. This explicitly loads the driver class. Since it does not depend on any external setup, this way of loading a driver is recommended. The following code loads the class `acme.db.Driver`:

   ```
   Class.forName("acme.db.Driver");
   ```

   If `acme.db.Driver` has been written so that loading it causes an instance to be created and also calls `DriverManager.registerDriver` with that instance as the parameter (as it should do), then it is in the `DriverManager`'s list of drivers and available for creating a connection.

2. By adding the driver to the `java.lang.System` property `jdbc.drivers`. This is a list of driver classnames, separated by colons, that the `DriverManager` class loads. When the `DriverManager` class is intialized, it looks for the system property `jdbc.drivers`, and if the user has entered one or more drivers, the `DriverManager` class attempts to load them. The following code illustrates how a programmer might enter three driver classes in `~/.hotjava/properties` (HotJava loads these into the system properties list on startup):

   ```
   jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.test.ourDriver;
   ```

The first call to a `DriverManager` method will automatically cause these driver classes to be loaded.

Note that this second way of loading drivers requires a preset environment that is persistent. If there is any doubt about that being the case, it is safer to call the method `Class.forName` to explicitly load each driver. This is also the method to use to bring in a particular driver since once the `DriverManager` class has been initialized, it will never recheck the `jdbc.drivers` property list.

In both of the cases listed above, it is the responsibility of the newly-loaded `Driver` class to register itself by calling `DriverManager.registerDriver`. As mentioned above, this should be done automatically when the class is loaded.

For security reasons, the JDBC management layer will keep track of which class loader provided which driver. Then when the `DriverManager` class is opening a connection, it will use only drivers that come from the local file system or from the same class loader as the code issuing the request for a connection.

3.1.2 Establishing a Connection

Once the `Driver` classes have been loaded and registered with the `DriverManager` class, they are available for establishing a connection with a database. When a request for a connection is made with a call to the `DriverManager.getConnection` method, the `DriverManager` tests each driver in turn to see if it can establish a connection.

It may sometimes be the case that more than one JDBC driver is capable of connecting to a given URL. For example, when connecting to a given remote database, it might be possible to use a JDBC-ODBC bridge driver, a JDBC-to-generic-network-protocol driver, or a driver supplied by the database vendor. In such cases, the order in which the drivers are tested is significant because the `DriverManager` will use the first driver it finds that can successfully connect to the given URL.

First the `DriverManager` tries to use each of the drivers in the order they were registered. (The drivers listed in `jdbc.drivers` are always registered first.) 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.

It tests the drivers by calling the method `Driver.connect` on each one in turn, passing them the URL that the user originally passed to the method `DriverManager.getConnection`. The first driver that recognizes the URL makes the connection.

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

The following code is an example of all that is normally needed to set up a connection with a driver such as a JDBC-ODBC bridge driver:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  //loads the driver
String url = "jdbc:odbc:fred";
DriverManager.getConnection(url, "userID", "passwd");
```

4 Statement

This overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

4.1 Overview

A `Statement` object is used to send SQL statements to a database. There are actually three kinds of `Statement` objects, all of which act as containers for executing SQL statements on a given connection: `Statement`, `PreparedStatement`, which inherits from `Statement`, and `CallableStatement`, which inherits from `PreparedStatement`. They are specialized for sending particular types of SQL statements: a `Statement` object is used to execute a simple SQL statement with no parameters; a `PreparedStatement` object is used to execute a precompiled SQL statement with or without IN parameters; and a `CallableStatement` object is used to execute a call to a database stored procedure.

The `Statement` interface provides basic methods for executing statements and retrieving results. The `PreparedStatement` interface adds methods for dealing with IN parameters; `CallableStatement` adds methods for dealing with OUT parameters.

4.1.1 Creating `Statement` Objects

Once a connection to a particular database is established, that connection can be used to send SQL statements. A `Statement` object is created with the `Connection` method `createStatement`, as in the following code fragment:

```
Connection con = DriverManager.getConnection(url, "sunny", "");
Statement stmt = con.createStatement();
```

The SQL statement that will be sent to the database is supplied as the argument to one of the methods for executing a `Statement` object:

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table2);
```

4.1.2 Executing `Statement` Objects

The `Statement` interface provides three different methods for executing SQL statements, `executeQuery`, `executeUpdate`, and `execute`. The one to use is determined by what the SQL statement produces.

The method `executeQuery` is designed for statements that produce a single result set. For the sake of clarity, we will distinguish between the terms *return value*, which is what the execution of a method returns, and *result*, which is what the SQL statement produces. For example, the method `executeQuery` returns a `ResultSet` object. The SQL statement that it executes produces a result set. So in this case, the return value (a `ResultSet` object which contains the result set generated by the SQL statement) is the same as the result.

The situation is different for the method `executeUpdate`, however. It is used to execute `INSERT`, `UPDATE`, or `DELETE` statements and also SQL DDL (Data Definition Language) statements like `CREATE TABLE` and `DROP TABLE`. The *result* of an `INSERT`, `UPDATE`, or `DELETE` statement is a modification of one or more columns in zero or more rows in a table. The *return value* of `executeUpdate` is an integer indicating the number of rows that were affected (referred to as the update count), which is quite different from the *results* of executing an update statement. A statement like `CREATE TABLE` presents yet another situation; it produces a new table, which is its result, but it returns nothing. In this case, the method `executeUpdate` returns zero. Consequently, when zero is the return value for `executeUpdate`, it can mean one of two things: 1) the SQL statement executed was an `INSERT`, `UPDATE`, or `DELETE` statement that affected no rows, or 2) the SQL statement executed was a DDL statement.

All of the methods for executing statements close the calling `Statement` object's current result set if there is one open. This means that one needs to complete any processing of the current `ResultSet` object before re-executing a `Statement` object.

It should be noted that the `PreparedStatement` interface, which inherits all of the methods in the `Statement` interface, has its own versions of the methods `executeQuery`, `executeUpdate` and `execute`. `Statement` objects do not themselves contain an SQL statement; therefore, one must be provided as the argument to the `Statement.execute` methods. `PreparedStatement` objects do not supply an SQL statement as a parameter to these methods because they already contain a precompiled SQL statement. `CallableStatement` objects inherit the `PreparedStatement` forms of these methods. Using a query parameter with `PrepredStatement` or `CallableStatement` versions of these methods will cause an `SQLException` to be thrown.

4.1.3 Using the Method `Execute`

The `execute` method should be used only when it is possible that a statement may return more than one `ResultSet` object, more than one update count, or a combination of `ResultSet` objects and update counts. These multiple possibilities for results, though rare, are possible when one is executing certain stored procedures or dynamically executing an unknown SQL string (that is, unknown to the application programmer at compile time). For example, a user might execute a stored procedure (using a `CallableStatement` object—see Section 7, "CallableStatement," of this *JDBC Guide*), and that stored procedure could perform an update, then a select, then an update, then a select, and so on. Typically someone using a stored procedure will know what it returns.

Because the method `execute` handles the cases that are out of the ordinary, it is no surprise that retrieving its results requires some special handling. For instance, suppose it is known that a procedure returns two result sets. After using the method `execute` to execute the procedure, one must call the method `getResultSet` to get the first result set and then the appropriate `getXXX` methods to retrieve values from it. To get the second result set, one needs to call `getMoreResults` and then `getResultSet` a second time. If it is known that a procedure returns an update count, the method `getUpdateCount` is called.

Those cases where one does not know what will be returned are more complicated. The method `execute` returns `true` if the result is a `ResultSet` object and `false` if it is a Java `int`. If it returns an `int`, that means that the result is either an update count or that the statement executed was a DDL command. The first thing to do after calling the method `execute`, is to call either `getResultSet` or `getUpdateCount`. The method `getResultSet` is called to get what might be the first of two or more `ResultSet` objects; the method `getUpdateCount` is called to get what might be the first of two or more update counts.

When the result of an SQL statement is not a result set, the method `getResultSet` will return `null`. This can mean that the result is an update count or that there are no more results. The only way to find out what the `null` really means in this case is to call the method `getUpdateCount`, which will return an integer. This integer will be the number of rows affected by the calling statement or `-1` to indicate either that the result is a result set or that there are no results. If the method `getResultSet` has already returned `null`, which means that the result is not a `ResultSet` object, then a return value of `-1` has to mean that there are no more results. In other words, there are no results (or no more results) when the following is true:

```
((stmt.getResultSet() == null) && (stmt.getUpdateCount() == -1))
```

If one has called the method `getResultSet` and processed the `ResultSet` object it returned, it is necessary to call the method `getMoreResults` to see if there is another result set or update count. If `getMoreResults` returns `true`, then one needs to again call `getResultSet` to actually retrieve the next result set. As already stated above, if `getResultSet` returns `null`, one has to call `getUpdateCount` to find out whether `null` means that the result is an update count or that there are no more results.

When `getMoreResults` returns `false`, it means that the SQL statement returned an update count or that there are no more results. So one needs to call the method `getUpdateCount` to find out which is the case. In this situation, there are no more results when the following is true:

```
((stmt.getMoreResults() == false) && (stmt.getUpdateCount() == -1))
```

The code below demonstrates one way to be sure that one has accessed all the result sets and update counts generated by a call to the method `execute`:

```
stmt.execute(queryStringWithUnknownResults);
while(true)  {
    int rowCount = stmt.getUpdateCount();
    if(rowCount > 0) {    //  this is an update count
        System.out.println("Rows changed = " + count);
        stmt.getMoreResults();
        continue;
```

```
        }
        if(rowCount = 0) {     // DDL command or 0 updates
            System.out.println(" No rows changed or statement was DDL
                                 command");
            stmt.getMoreResults();
            continue;
        }

    // if we have gotten this far, we have either a result set
    // or no more results

        ResultSet rs = stmt.getResultSet;
        if(rs != null) {
            . . .  // use metadata to get info about result set columns
            while(rs.next())  {
            . . .  // process results
            stmt.getMoreResults();
            continue;
        }
        break;       // there are no more results
    }
```

4.1.4 `Statement` Completion

When a connection is in auto-commit mode, the statements being executed within it are committed or rolled back when they are completed. A statement is considered complete when it has been executed and all its results have been returned. For almost all cases, this occurs when one calls the method `executeQuery` (and retrieves the `ResultSet` rows) or the method `executeUpdate`. In the rare cases where the method `execute` is called, however, a statement is not complete until all of the result sets or update counts it generated have been retrieved.

Some DBMSs treat each statement in a stored procedure as a separate statement; others treat the entire procedure as one compound statement. This difference becomes important when auto-commit is enabled because it affects when the method `commit` is called. In the first case, each statement is individually committed; in the second, all are committed together.

4.1.5 SQL Escape Syntax in `Statement` Objects

`Statement` objects may contain SQL statements that use SQL escape syntax. Escape syntax signals the driver that the code within it should be handled differently. The driver will scan for any escape syntax and translate it into code that the particular database understands. This makes escape syntax DBMS-independent and allows a programmer to use features that might not otherwise be available.

An escape clause is demarcated by curly braces and a key word:

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

The keyword indicates the kind of escape clause, as shown below.

- `escape` for `LIKE` escape characters

  The characters "%" and "_" work like wild cards in SQL `LIKE` clauses ("%" matches zero or more characters, and "_" matches exactly one character). In order to interpret them literally, they can be preceded by a backslash ("\"), which is a special escape character in strings. One can specify which character to use as the escape character by including the following syntax at the end of a query:

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

  For example, the following query, using the backslash character as an escape character, finds identifier names that begin with an underbar:

```
stmt.executeQuery("SELECT name FROM Identifiers
        WHERE Id LIKE '\_%' {escape '\'};
```

- `fn` for scalar functions

  Almost all DBMSs have numeric, string, time, date, system, and conversion functions on scalar values. One of these functions can be used by putting it in escape syntax with the keyword `fn` followed by the name of the desired function and its arguments. For example, the following code calls the function `concat` with two arguments to be concatenated:

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

The name of the current database user can be obtained with the following syntax:

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

Scalar functions may be supported by different DBMSs with slightly different syntax, and they may not be supported by all drivers. Various `DatabaseMetaData` methods will list the functions that are supported. For example, the method `getNumericFunctions` returns a comma-separated list of the names of numeric functions, the method `getStringFunctions` returns string functions, and so on.

The driver will either map the escaped function call into the appropriate syntax or implement the function directly itself.

- `d`, `t`, and `ts` for date and time 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 following syntax:

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

In this syntax, `yyyy` is the year, `mm` is the month, and `dd` is the day. The driver will replace the escape clause with the equivalent DBMS-specific representation. For example, the driver might replace `{d 1999-02-28}` with `'28-FEB-99'` if that is the appropriate format for the underlying database.

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.

- `call` or `? = call` for stored procedures

If a database supports stored procedures, they can be invoked from JDBC with the following syntax:

```
{call procedure_name[(?, ?, . . .)]}
```

or, where a procedure returns a result parameter:

```
{? = call procedure_name[(?, ?, . . .)]}
```

The square brackets indicate that the material enclosed between them is optional. They are not part of the syntax.

Input arguments may be either literals or parameters. See Section 7, “CallableStatement,” of this *JDBC Guide* for more information.

One can call the method `DatabaseMetaData.supportsStoredProcedures` to see if the database supports stored procedures.

- `oj` for 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
```

Outer joins are an advanced feature, and one can check the SQL grammar for an explanation of them. JDBC provides three `DatabaseMetaData` methods for determining the kinds of outer joins a driver supports: `supportsOuterJoins`, `supportsFullOuterJoins`, and `supportsLimitedOuterJoins`.

The method `Statement.setEscapeProcessing` turns escape processing on or off; the default is for it to be on. A programmer might turn it off to cut down on processing time when performance is paramount, but it would normally be turned on. It should be noted that `setEscapeProcessing` does not work for `PreparedStatement` objects because the statement may have already been sent to the database before it can be called. See `PreparedStatement` regarding precompilation.

5 ResultSet

THIS overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

5.1 Overview

A `ResultSet` contains all of the rows which satisfied the conditions in an SQL statement, and it provides access to the data in those rows through a set of `get` methods that allow access to the various columns of the current row. The `ResultSet.next` method is used to move to the next row of the `ResultSet`, making the next row become the current row.

The general form of a result set is a table with column headings and the corresponding values returned by a query. For example, if your query is `SELECT a, b, c FROM Table1`, your result set will have the following form:

```
a          b            c
--------   ---------    --------
12345      Cupertino    CA
83472      Redmond      WA
83492      Boston       MA
```

The following code fragment is an example of executing an 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");
    float f = r.getFloat("c");
    System.out.println("ROW = " + i + " " + s + " " + f);
}
```

5.1.1 Rows and Cursors

A `ResultSet` maintains a cursor which points to its current row of data. The cursor moves down one row each time the method `next` is called. Initially it is positioned before the first row, so that the first call to `next` puts the cursor on the first row, making it the current row. `ResultSet` rows are retrieved in sequence from the top row down as the cursor moves down one row with each successive call to `next`.

A cursor remains valid until the `ResultSet` object or its parent `Statement` object is closed.

In SQL, the cursor for a result table is named. If a database allows positioned updates or positioned deletes, the name of the cursor needs to be supplied as a parameter to the update or delete command. This cursor name can be obtained by calling the method `getCursorName`.

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 ensure that rows selected are properly locked so that positioned updates do not result in update anomalies or other concurrency problems.

5.1.2 Columns

The `getXXX` methods provide the means for retrieving column values from the current row. Within each row, column values may be retrieved in any order, but for maximum portability, one should retrieve values from left to right and read column values only once.

Either the column name or the column number can be used to designate the column from which to retrieve data. For example, if the second column of a `ResultSet` object *`rs`* is named “title” and stores values as strings, either of the following will retrieve the value stored in that column:

```
String s = rs.getString("title");
String s = rs.getString(2);
```

Note that columns are numbered from left to right starting with column 1. Also, column names used as input to `getXXX` methods are case insensitive.

The option of using the column name was provided so that a user who specifies column names in a query can use those same names as the arguments to `getXXX` methods. If, on the other hand, the `select` statement does not specify column names (as in "`select * from table1`" or in cases where a column is derived), column numbers should be used. In such situations, there is no way for the user to know for sure what the column names are.

In some cases, it is possible for a SQL query to return a result set that has more than one column with the same name. If a column name is used as the parameter to a `getXXX` method, `getXXX` will return the value of the first matching column name. Thus, if there are multi0ple columns with the same name, one needs to use a column index to be sure that the correct column value is retrieved. It may also be slightly more efficient to use column numbers.

Information about the columns in a `ResultSet` is available by calling the method `ResultSet.getMetaData`. The `ResultSetMetaData` object returned gives the number, types, and properties of its `ResultSet` object's columns.

If the name of a column is known, but not its index, the method `findColumn` can be used to find the column number.

5.1.3 Data Types and Conversions

For the `getXXX` methods, the JDBC driver attempts to convert the underlying data to the specified Java type and then returns a suitable Java value. For example, if the `getXXX` method is `getString`, and the data type of the data in the underlying database is `VARCHAR`, the JDBC driver will convert `VARCHAR` to Java `String`. The return value of `getString` will be a Java `String` object.

The following table shows which SQL types a `getXXX` method is *allowed* to retrieve and which SQL types are *recommended* for it to retrieve. A small `x` indicates a legal `getXXX` method for a particular data type; a large `X` indicates the recommended `getXXX` method for a data type. For example, any `getXXX` method except `getBytes` or `getBinaryStream` can be used to retrieve the value of a `LONGVARCHAR`, but `getAsciiStream` or `getUnicodeStream` are recommended, depending on which data type is being returned. The method `getObject` will return any data type as a Java `Object` and is useful when the underlying data type is a database-specific abstract type or when a generic application needs to be able to accept any data type.

Use of `ResultSet.getXXX` methods to retrieve common SQL data types.

An "x" indicates that the `getXXX` method may legally be used to retrieve the given SQL type.
An "X" indicates that the `getXXX` method is recommended for retrieving the given SQL type.

	TINYINT	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						
getBigDecimal	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**			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

5.1.4 Using Streams for Very Large Row Values

`ResultSet` makes it possible to retrieve arbitrarily large `LONGVARBINARY` or `LONGVARCHAR` data. The methods `getBytes` and `getString` return data as one large chunk (up to the limits imposed by the return value of `Statement.getMaxFieldSize`). However, it may be more convenient to retrieve very large data in smaller, fixed-size chunks. This is done by having the `ResultSet` class return `java.io.Input` streams from which data can be read in chunks. Note that these streams must be accessed immediately because they will be closed automatically on the next `getXXX` call on `ResultSet`. (This behavior is imposed by underlying implementation constraints on large blob access.)

The JDBC API has three separate methods for getting streams, each with a different return value:

- `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 two-byte Unicode characters.

Note that this differs from Java streams, which return untyped bytes and can (for example) be used for both ASCII and Unicode characters.

The following code gives an example of using `getAsciiStream`:

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

```
}
```

5.1.5 NULL Result Values

To determine if a given result value is SQL NULL, one must first read the column and then use the `ResultSet.wasNull` method to discover if the read returned an SQL `NULL`.

When one has read an SQL `NULL` using one of the `ResultSet.getXXX` methods, the method `wasNull` will return one of the following:

- A Java `null` value for those `getXXX` methods that return Java objects (methods such as `getString`, `getBigDecimal`, `getBytes`, `getDate`, `getTime`, `getTimestamp`, `getAsciiStream`, `getUnicodeStream`, `getBinaryStream`, `getObject`).
- A zero value for `getByte`, `getShort`, `getInt`, `getLong`, `getFloat`, and `getDouble`.
- A `false` value for `getBoolean`.

5.1.6 Optional or Multiple Result Sets

Normally SQL statements are 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 result set until the statement has executed. In addition, some stored procedures may return several different result sets and/or update counts.

To accommodate these situations, JDBC provides a mechanism so that an application can execute a statement and then process an arbitrary collection of result sets and update counts. This mechanism is based on first calling a fully general `execute` method, and then calling three other methods, `getResultSet`, `getUpdateCount`, and `getMoreResults`. 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.

You do not need to do anything to close a `ResultSet`; it is automatically closed by the `Statement` that generated it when that `Statement` is closed, is re-executed, or is used to retrieve the next result from a sequence of multiple results.

6 PreparedStatement

THIS overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

6.1 Overview

The `PreparedStatement` interface inherits from `Statement` and differs from it in two ways:

1. Instances of `PreparedStatement` contain an SQL statement that has already been compiled. This is what makes a statement "prepared."
2. The SQL statement contained in a `PreparedStatement` object may have one or more IN parameters. An IN parameter is a parameter whose value is not specified when the SQL statement is created. Instead the statement has a question mark ("?") as a placeholder for each IN parameter. A value for each question mark must be supplied by the appropriate `setXXX` method before the statement is executed.

Because `PreparedStatement` objects are precompiled, their execution can be faster than that of `Statement` objects. Consequently, an SQL statement that is executed many times is often created as a `PreparedStatement` object to increase efficiency.

Being a subclass of `Statement`, `PreparedStatement` inherits all the functionality of `Statement`. In addition, it adds a whole set of methods which are needed for setting the values to be sent to the database in place of the placeholders for IN parameters. Also, the three methods `execute`, `executeQuery`, and `executeUpdate` are modified so that they take no argument. The `Statement` forms of these methods (the forms that take an SQL statement parameter) should never be used with a `PreparedStatement` object.

6.1.1 Creating `PreparedStatement` Objects

The following code fragment, where `con` is a `Connection` object, creates a `PreparedStatement` object containing an SQL statement with two placeholders for IN parameters:

```
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE table4 SET m = ? WHERE x = ?");
```

The object `pstmt` now contains the statement `"UPDATE table4 SET m = ? WHERE x = ?"`, which has already been sent to the DBMS and been prepared for execution.

6.1.2 Passing `IN` Parameters

Before a `PreparedStatement` object is executed, the value of each `?` parameter must be set. This is done by calling a `setXXX` method, where `XXX` is the appropriate type for the parameter. For example, if the parameter has a Java type of `long`, the method to use is `setLong`. The first argument to the `setXXX` methods is the *ordinal position* of the parameter to be set, and the second argument is the *value* to which the parameter is to be set. For example, the following code sets the first parameter to `123456789` and the second parameter to `100000000`:

```
pstmt.setLong(1, 123456789);
pstmt.setLong(2, 100000000);
```

Once a parameter value has been set for a given statement, it can be used for multiple executions of that statement until it is cleared by a call to the method `clearParameters`.

In the default mode for a connection (auto-commit enabled), each statement is commited or rolled back automatically when it is completed.

The same `PreparedStatement` object may be executed multiple times if the underlying database and driver will keep statements open after they have been committed. Unless this is the case, however, there is no point in trying to improve performance by using a `PreparedStatement` object in place of a `Statement` object.

Using `pstmt`, the `PreparedStatement` object created above, the following code illustrates setting values for the two parameter placeholders and executing `pstmt` 10 times. As stated above, for this to work, the database must not close `pstmt`. In this example, the first parameter is set to `"Hi"` and remains constant. The second parameter is set to a different value each time around the `for` loop, starting with `0` and ending with `9`.

```
pstmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
    pstmt.setInt(2, i);
    int rowCount = pstmt.executeUpdate();
}
```

6.1.3 Data Type Conformance on IN Parameters

The `XXX` in a `setXXX` method is a Java type. It is implicitly an SQL type because the driver will map the Java type to its corresponding SQL type (following the mapping specified in the table in Section 8.5.2 of "Mapping Java and SQL Types" in this *JDBC Guide*) and send that SQL type to the database. For example, the following code fragment sets the second parameter of the `PreparedStatement` object *`pstmt`* to `44`, with a Java type of `short`:

```
pstmt.setShort(2, 44);
```

The driver will send 44 to the database as an SQL `SMALLINT`, which is the standard mapping from a Java `short`.

It is the programmer's responsibility to make sure that the Java type of each IN parameter maps to an SQL type that is compatible with the SQL data type expected by the database. Consider the case where the database expects an SQL `SMALLINT`. If the method `setByte` is used, the driver will send an SQL `TINYINT` to the database. This will probably work because many databases convert from one related type to another, and generally a `TINYINT` can be used anywhere a `SMALLINT` is used. However, for an application to work with the most databases possible, it is best to use Java types that correspond to the exact SQL types expected by the database. If the expected SQL type is `SMALLINT`, using `setShort` instead of `setByte` will make an application more portable.

6.1.4 Using `setObject`

A programmer can explicitly convert an input parameter to a particular SQL type by using the method `setObject`. This method can take a third argument, which specifies the target SQL type. The driver will convert the Java `Object` to the specified SQL type before sending it to the database.

If no SQL type is given, the driver will simply map the Java `Object` to its default SQL type (using the table in Section 8.5.4) and then send it to the database. This is similar to what happens with the regular `setXXX` methods; in both cases, the driver maps the Java type of the value to the appropriate SQL type before sending it to the database. The difference is that the `setXXX` methods use the standard mapping from Java types to SQL types (see the table in Section 8.5.2), whereas the `setObject` method uses the mapping from Java `Object` types to SQL types (see the table in Section 8.5.4).

The capability of the method `setObject` to accept any Java object allows an application to be generic and accept input for a parameter at run time. In this situation the type of the input is not known when the application is compiled. By using `setObject`, the application can accept any Java object type as input and convert it to the SQL type expected by the database. The table in Section 8.5.5 shows all the possible conversions that `setObject` can perform.

6.1.5 Sending `SQL NULL` as an IN parameter

The `setNull` method allows a programmer to send an SQL `NULL` value to the database as an IN parameter. Note, however, that one must still specify the SQL type of the parameter.

An SQL `NULL` will also be sent to the database when a Java `null` value is passed to a `setXXX` method (if it takes Java objects as arguments). The method `setObject`, however, can take a `null` value only if the SQL type is specified.

6.1.6 Sending Very Large IN Parameters

The methods `setBytes` and `setString` are capable of sending unlimited amounts of data. Sometimes, however, programmers prefer to pass in large blobs of data in smaller chunks. This can be accomplished by setting an IN parameter to a Java input stream. When the statement is executed, the JDBC driver will make repeated calls to this input stream, reading its contents and transmitting those contents as the actual parameter data.

JDBC provides three methods for setting IN parameters to input streams: `setBinaryStream` for streams containing uninterpreted bytes, `setAsciiStream` for streams containing ASCII characters, and `setUnicodeStream` for streams containing Unicode characters. These methods take one more argument than the other `setXXX` methods because the total length of the stream must be specified. This is necessary because some databases need to know the total transfer size before any data is sent.

The following code illustrates using a stream to send the contents of a file as an IN parameter:

```
java.io.File file = new java.io.File("/tmp/data");
int fileLength = file.length();
java.io.InputStream fin = new java.io.FileInputStream(file);
java.sql.PreparedStatement pstmt = con.prepareStatement(
   "UPDATE Table5 SET stuff = ? WHERE index = 4");
pstmt.setBinaryStream (1, fin, fileLength);
pstmt.executeUpdate();
```

When the statement executes, the input stream `fin` will get called repeatedly to deliver up its data.

7 CallableStatement

THIS overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

7.1 Overview

A `CallableStatement` object provides a way to call stored procedures in a standard way for all DBMSs. A stored procedure is stored in a database; the *call* to the stored procedure is what a `CallableStatement` object contains. This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one. (See Section 4, "Statement," for information on escape syntax.) A result parameter, a kind of OUT parameter, is the return value for the stored procedure. Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters). A question mark serves as a placeholder for a parameter.

The syntax for invoking a stored procedure in JDBC is shown below. Note that the square brackets indicate that what is between them is optional; they are not themselves part of the syntax.

```
{call procedure_name[(?, ?, ...)]}
```

The syntax for a procedure that returns a result parameter is:

```
{? = call procedure_name[(?, ?, ...)]}
```

The syntax for a stored procedure with no parameters would look like this:

```
{call procedure_name}
```

Normally, anyone creating a `CallableStatement` object would already know that the DBMS being used supports stored procedures and what those procedures are. If one needed to check, however, various `DatabaseMetaData` methods will supply such information. For instance, the method `supportsStoredProcedures` will return `true` if the DBMS supports stored procedure calls, and the method `getProcedures` will return a description of the stored procedures available.

`CallableStatement` inherits `Statement` methods, which deal with SQL statements in general, and it also inherits `PreparedStatement` methods, which deal with IN parameters. All of the methods defined in `CallableStatement` deal with OUT parameters or the output aspect of INOUT parameters: registering the SQL types of the OUT parameters, retrieving values from them, or checking whether a returned value was `SQL NULL`.

7.1.1 Creating a `CallableStatement` Object

`CallableStatement` objects are created with the `Connection` method `prepareCall`. The example below creates an instance of `CallableStatement` that contains a call to the stored procedure `getTestData`, which has two arguments and no result parameter:

```
CallableStatement cstmt = con.prepareCall(
                     "{call getTestData(?, ?)}");
```

Whether the `?` placeholders are IN, OUT, or INOUT parameters depends on the stored procedure `getTestData`.

7.1.2 IN and OUT Parameters

Passing in any IN parameter values to a `CallableStatement` object is done using the `setXXX` methods inherited from `PreparedStatement`. The type of the value being passed in determines which `setXXX` method to use (`setFloat` to pass in a `float` value, and so on).

If the stored procedure returns OUT parameters, the SQL type of each OUT parameter must be registered before the `CallableStatement` object can be executed. (This is necessary because some DBMSs require the SQL type.) Registering the SQL type is done with the method `registerOutParameter`. Then after the statement has been executed, `CallableStatement`'s `getXXX` methods retrieve the parameter value. The correct `getXXX` method to use is the Java type that corresponds to the SQL type registered for that parameter. (The standard mapping from SQL types to Java types is shown in the table in Section 8.5.1.) In other words, `registerOutParameter` uses an SQL type (so that it matches the SQL type that the database will return), and `getXXX` casts this to a Java type.

To illustrate, the following code registers the OUT parameters, executes the stored procedure called by *`cstmt`*, and then retrieves the values returned in the OUT parameters. The method `getByte` retrieves a Java byte from the first OUT parameter, and `getBigDecimal` retrieves a `BigDecimal` object (with three digits after the decimal point) from the second OUT parameter:

```
CallableStatement cstmt = con.prepareCall(
                                        "{call getTestData(?, ?)}");
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.registerOutParameter(2, java.sql.Types.DECIMAL, 3);
cstmt.executeQuery();
byte x = cstmt.getByte(1);
java.math.BigDecimal n = cstmt.getBigDecimal(2, 3);
```

Unlike `ResultSet`, `CallableStatement` does not provide a special mechanism for retrieving large OUT values incrementally.

7.1.3 INOUT Parameters

A parameter that supplies input as well as accepts output (an INOUT parameter) requires a call to the appropriate `setXXX` method (inherited from `PreparedStatement`) in addition to a call to the method `registerOutParameter`. The `setXXX` method sets a parameter's value as an input parameter, and the method `registerOutParameter` registers its SQL type as an output parameter. The `setXXX` method provides a Java value which the driver converts to an SQL value before sending it to the database. The SQL type of this IN value and the SQL type supplied to the method `registerOutParameter` should be the same. Then to retrieve the output value, a corresponding `getXXX` method is used. For example, a parameter whose Java type is `byte` should use the method `setByte` to assign the input value, should supply a `TINYINT` as the SQL type to `registerOutParameter`, and should use `getByte` to retrieve the output value. (Section 8, "Mapping SQL and Java Types," gives more information and contains tables of type mappings.)

The following example assumes that there is a stored procedure `reviseTotal` whose only parameter is an INOUT parameter. The method `setByte` sets the parameter to `25`, which the driver will send to the database as an SQL `TINYINT`. Next `registerOutParameter` registers the parameter as an SQL `TINYINT`. After the stored procedure is executed, a new SQL `TINYINT` value is returned, and the method `getByte` will retrieve this new value as a Java `byte`.

```
CallableStatement cstmt = con.prepareCall(
        "{call reviseTotal(?)}");
cstmt.setByte(1, 25);
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.executeUpdate();
byte x = cstmt.getByte(1);
```

7.1.4 Retrieve OUT Parameters after Results

Because of limitations imposed by some DBMSs, it is recommended that for maximum portability, all of the results generated by the execution of a `CallableStatement` object should be retrieved before OUT parameters are retrieved using `CallableStatement.getXXX` methods.

If a `CallableStatement` object returns multiple `ResultSet` objects (using a call to the method `execute`), all of the results should be retrieved before OUT parameters are retrieved. In this case, to be sure that all results have been accessed, the `Statement` methods `getResultSet`, `getUpdateCount`, and `getMoreResults` need to be called until there are no more results.

After this is done, values from OUT parameters can be retrieved using the `CallableStatement.getXXX` methods.

7.1.5 Retrieving NULL Values as OUT Parameters

The value returned to an OUT parameter may be `SQL NULL`. When this happens, the `SQL NULL` value will be converted so that the value returned by a `getXXX` method will be `null`, `0`, or `false`, depending on the `getXXX` method type. As with `ResultSet` objects, the only way to know if a value of `0` or `false` was originally `SQL NULL` is to test it with the method `wasNull`, which returns `true` if the last value read by a `getXXX` method was `SQL NULL` and `false` otherwise. Section 5, "ResultSet," contains more information.

8
Mapping SQL and Java Types

THIS overview is excerpted from *JDBC™ Database Access with Java™: A Tutorial and Annotated Reference,* currently in progress at JavaSoft. This book, both a tutorial and the definitive reference manual for JDBC, will be published in the spring of 1997 by Addison-Wesley Publishing Company as part of the Java series.

8.1 Overview

Since SQL data types and Java data types are not identical, there needs to be some mechanism for reading and writing data between an application using Java types and a database using SQL types.

To accomplish this, JDBC provides sets of `getXXX` and `setXXX` methods, the method `registerOutParameter`, and the class `Types`.

This section brings together information about data types affecting various classes and interfaces and puts all the tables showing the mappings between SQL types and Java types in one place for easy reference.

8.2 Mapping SQL Data Types into Java

JDBC provides a standard mapping from the common SQL data types to Java types. For example, an SQL `INTEGER` is normally mapped to a Java `int`. This supports a simple interface for reading and writing SQL values as simple Java types.

The Java types do not need to be exactly isomorphic to the SQL types; they just need to be able to represent them with enough type information to correctly store and retrieve parameters and recover results from SQL statements. For example, a Java `String` object does not precisely match any of the `SQL CHAR` types, but it gives enough type information to represent `CHAR`, `VARCHAR`, or `LONGVARCHAR` successfully.

8.3 SQL Types

This section describes the common SQL types and how they are mapped to Java types.

8.3.1 `CHAR`, `VARCHAR`, and `LONGVARCHAR`

Java programmers do not need to distinguish among the three types of SQL strings, `CHAR`, `VARCHAR`, and `LONGVARCHAR`. Each can be expressed as a Java `String`, and it is possible to read and write an SQL statement correctly without knowing the exact data type that was expected.

`CHAR`, `VARCHAR`, and `LONGVARCHAR` could have been mapped to either `String` or `char[]`, but `String` is more appropriate for normal use. Also, the `String` class makes it easy to convert between `String` and `char[]`. There is a method for converting a `String` object to a `char[]` and also a constructor for turning a `char[]` into a `String` object.

One issue that had to be addressed is how to handle fixed-length SQL strings of type `CHAR(n)`. The answer is that JDBC drivers (or the DBMS) perform appropriate padding with spaces. Thus, when a `CHAR(n)` field is retrieved from the database, the driver will convert it to a Java `String` object of length `n`, which may include some padding spaces at the end. Conversely, when a `String` object 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 method `ResultSet.getString`, which allocates and returns a new `String` object, is recommended for retrieving data from `CHAR`, `VARCHAR`, and `LONGVARCHAR` fields. This is suitable for retrieving normal data, but can be unwieldy if the type `SQL LONGVARCHAR` is being used to store multimegabyte strings. To handle this case, two methods in the `ResultSet` interface allow programmers to retrieve a `LONGVARCHAR` value as a Java input stream from which they can subsequently read data in whatever size chunks they prefer. These methods are `getAsciiStream` and `getUnicodeStream`, which deliver the data stored in a `LONGVARCHAR` column as a stream of ASCII or Unicode characters.

8.3.2 `DECIMAL` and `NUMERIC`

The SQL data types `DECIMAL` and `NUMERIC`, used to express fixed-point numbers where absolute precision is required, can be expressed identically in Java. They are mapped to `java.math.BigDecimal`, a Java type that also expresses fixed-point numbers with absolute precision. The `java.math.BigDecimal` type provides math operations to allow `BigDecimal` types to be added, subtracted, multiplied, and divided with other `BigDecimal` types, with integer types, and with floating point types.

The method recommended for retrieving SQL `DECIMAL` and SQL `NUMERIC` values is `ResultSet.getBigDecimal`. JDBC also allows access to these SQL types as simple `Strings` or arrays of `char`. Thus, Java programmers can use `getString` to receive a `NUMERIC` or `DECIMAL` result. However, this makes the common case where `NUMERIC` or `DECIMAL` are used for currency values rather awkward, since it means that application writers have to perform math on strings. It is also possible to retrieve these SQL types as any of the Java numeric types.

8.3.3 `BINARY`, `VARBINARY`, and `LONGVARBINARY`

The SQL data types include three versions of raw binary values: `BINARY`, `VARBINARY`, and `LONGVARBINARY`. They can all be expressed identically as `byte` arrays in Java. Since it is possible to read and write SQL statements correctly without knowing the exact `BINARY` data type that was expected, there is no need for Java programmers to distinguish among them.

The method recommended for retrieving `BINARY` and `VARBINARY` values is `ResultSet.getBytes`. If a column of type SQL `LONGVARBINARY` stores a byte array that is many megabytes long, however, the method `getBinaryStream` is recommended. Similar to the situation with `LONGVARCHAR`, this method allows a Java programmer to retrieve a `LONGVARBINARY` value as a Java input stream that can be read later in smaller chunks.

8.3.4 BIT

The SQL type `BIT` is mapped directly to the Java type `boolean`.

8.3.5 `TINYINT`, `SMALLINT`, `INTEGER`, and `BIGINT`

The SQL types `TINYINT`, `SMALLINT`, `INTEGER`, and `BIGINT` are mapped as follows:

SQL `TINYINT` represents 8-bit values and is mapped to Java `byte`.
SQL `SMALLINT` represents 16-bit values and is mapped to Java `short`.
SQL `INTEGER` represents 32-bit values and is mapped to Java `int`.
SQL `BIGINT` represents 64-bit values and is mapped to Java `long`.

8.3.6 `REAL`, `FLOAT`, and `DOUBLE`

SQL defines three floating-point data types: `REAL`, `FLOAT`, and `DOUBLE`; whereas Java defines two: `FLOAT` and `DOUBLE`.

> `SQL REAL` is required to support 7 digits of mantissa precision and is mapped to Java `float`.
> `SQL FLOAT` and `SQL DOUBLE` are required to support 15 digits of mantissa precision and are mapped to Java `double`.

8.3.7 `DATE`, `TIME`, and `TIMESTAMP`

There are three SQL types relating to time:

- `DATE` consists of day, month, and year.
- `TIME` consists of of hours, minutes, and seconds.
- `TIMESTAMP` consists of `DATE` plus `TIME` plus a nanosecond field.

Because the standard Java class `java.util.Date` does not match any of these three SQL types exactly (it includes both `DATE` and `TIME` information but has no nanoseconds), JDBC defines three subclasses of `java.util.Date` to correspond to the SQL types. They are:

- `java.sql.Date` for `SQL DATE` information. The hour, minute, second, and millisecond fields of the `java.util.Date` base class are set to zero.
- `java.sql.Time` for `SQL TIME` information. The year, month, and day fields of the `java.util.Date` base class are set to 1970, January, and 1. This is the "zero" date in the Java epoch.
- `java.sql.Timestamp` for `SQL TIMESTAMP` information. This class extends `java.util.Date` by adding a nanosecond field.

All three of the JDBC time-related classes are subclasses of `java.util.Date`, and as such, they can be used where a `java.util.Date` is expected. For example, internationalization methods take a `java.util.Date` object as an argument, so they can be passed instances of any of the JDBC time-related classes.

A JDBC `Timestamp` object has its parent's date and time components and also a separate nanoseconds component. If a `java.sql.Timestamp` object is used where a `java.util.Date` object is expected, the nanoseconds component is lost.

However, since a `java.util.Date` object is stored with a precision of one millisecond, it is possible to maintain this degree of precision when converting a `java.sql.Timestamp` object to a `java.util.Date` object. This is done by converting the nanoseconds in the nanoseconds component to whole milliseconds (by dividing the number of nanoseconds by 1,000,000) and then adding the result to the the `java.util.Date` object. Up to 999,999 nanoseconds may be lost in this conversion, but the resulting `java.util.Date` object will be accurate to within one millisecond.

8.4 Examples of Mapping

In any situation where a Java program retrieves data from a database, there has to be some form of mapping and data conversion. In most cases, JDBC programmers will be programming with knowledge of their target database's schema. They would know, for example, what tables the database contains and the data type for each column in those tables. They can therefore use the strongly-typed access methods in the interfaces `ResultSet`, `PreparedStatement`, and `CallableStatement`. This section presents three different scenarios, describing the data mapping and conversion required in each.

8.4.1 Simple SQL Statement

In the most common case, a user executes a simple SQL statement and gets back a `ResultSet` object with the results. The value returned by the database and stored in a `ResultSet` column will have an SQL data type. A call to a `ResultSet.getXXX` method will retrieve that value as a Java data type. For example, if a `ResultSet` column contains an SQL `FLOAT` value, the method `getDouble` will retrieve that value as a Java `double`. The table in Section 8.5.6 shows which `getXXX` methods may be used to retrieve which SQL types. (A user who does not know the type of a `ResultSet` column can get that information by calling the method `ResultSet.getMetaData` and then invoking the `ResultSetMetaData` methods `getColumnType` or `getColumnTypeName`.) The following code fragment demonstrates getting the column type names for the columns in a result set:

```
String query = "select * from Table1";
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
```

```
for (int i = 1; i <= columnCount; i++)  {
    String s = rsmd.getColumnTypeName(i);
    System.out.println ("Column " + i + " is type " + s);
}
```

8.4.2 SQL Statement with IN Parameters

In another possible scenario, the user sends an SQL statement which takes input parameters. In this case, the user calls the `PreparedStatement.setXXX` methods to assign a value to each input parameter. For example, `PreparedStatement.setLong(1, 2345678)` will assign the value `2345678` to the first parameter as a Java `long`. The driver will convert `2345678` to an SQL `BIGINT` in order to send it to the database. Which SQL type the driver sends to the database is determined by the standard mapping from Java types to SQL types, which is shown in the table in Section 8.5.2.

8.4.3 SQL Statement with INOUT Parameters

In yet another scenario, a user wants to call a stored procedure, assign values to its INOUT parameters, retrieve values from the results, and retrieve values from the parameters. This case is rather uncommon and more complicated than most, but it gives a good illustration of mapping and data conversion.

In this scenario, the first thing to do is to assign values to the INOUT parameters using `PreparedStatement.setXXX` methods. In addition, since the parameters will also be used for output, the programmer must register each parameter with the SQL type of the value that the database will return to it. This is done with the method `CallableStatement.registerOutParameter`, which takes one of the SQL types defined in the class `Types`. A programmer retrieves the results returned to a `ResultSet` object with `ResultSet.getXXX` methods and retrieves the values stored in the output parameters with `CallableStatement.getXXX` methods.

The `XXX` type used for `ResultSet.getXXX` methods is fairly flexible in some cases. The table in Section 8.5.6 shows which `ResultSet.getXXX` methods can be used to retrieve which SQL types.

The `XXX` type used for `CallableStatement.getXXX` must map to the SQL type registered for that parameter. For example, if the database is expected to return an output value whose type is `SQL REAL`, the parameter should have been registered as `java.sql.Types.REAL`. Then to retrieve the `SQL REAL` value, the method `CallableStatement.getFloat` should be called (the mapping from SQL types to Java types is shown in the table in Section 8.5.1). The method `getFloat` will return the value stored in the output parameter after converting it from an SQL `REAL` to a Java `float`. To accommodate various databases and make an application more portable, it is recommended that values be retrieved from `ResultSet` objects before values are retrieved from output parameters.

The following code demonstrates calling a stored procedure named `getTestData`, which has two parameters that are both INOUT parameters. First the `Connection` object *`con`* creates the `CallableStatement` object *`cstmt`*. Then the method `setByte` sets the first parameter to `25` as a Java `byte`. The driver will convert `25` to an SQL `TINYINT` and send it to the database. The method `setBigDecimal` sets the second parameter with an input value of `83.75`. The driver will convert this `java.math.BigDecimal` object to an SQL `NUMERIC` value. Next the two parameters are registered as OUT parameters, the first parameter as an SQL `TINYINT` and the second parameter as an SQL `DECIMAL` with two digits after the decimal point. After *`cstmt`* is executed, the values are retrieved from the `ResultSet` object using `ResultSet.getXXX` methods. The method `getString` gets the value in the first column as a Java `String` object, `getInt` gets the value in the second column as a Java `int`, and `getInt` gets the value in the third column as a Java `int`.

Then `CallableStatement.getXXX` methods retrieve the values stored in the output parameters. The method `getByte` retrieves the `SQL TINYINT` as a Java `byte`, and `getBigDecimal` retrieves the `SQL DECIMAL` as a `java.math.BigDecimal` object with two digits after the decimal point. Note that when a parameter is both an input and an output parameter, the `setXXX` method uses the same Java type as the `getXXX` method (as in `setByte` and `getByte`). The `registerOutParameter` method registers it to the SQL type that is mapped from the Java type (a Java `byte` maps to an SQL `TINYINT`, as shown in the table in Section 8.5.2).

```
CallableStatement cstmt = con.prepareCall(
          "{call getTestData(?, ?)}");
cstmt.setByte(1, 25);
cstmt.setBigDecimal(2, 83.75);
// register the first parameter as an SQL TINYINT and the second
//parameter as an SQL DECIMAL with two digits after the decimal point
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.registerOutParameter(2, java.sql.Types.DECIMAL, 2);
ResultSet rs = cstmt.executeUpdate();
// retrieve and print values in result set
while(rs.next()) {
    String name = rs.getString(1);
    int score = rs.getInt(2);
    int percentile = rs.getInt(3);
```

```
    System.out.print("name = " + name + ", score = " + score + ", "
    System.out.println("percentile = " + percentile);
// retrieve values in output parameters
byte x = cstmt.getByte(1);
java.math.BigDecimal n = cstmt.getBigDecimal(2, 2);
```

To generalize, the `XXX` in `CallableStatement.getXXX` and `PreparedStatement.setXXX` methods is a Java type. For `setXXX` methods, the driver converts the Java type to an SQL type before sending it to the database (using the standard mappings shown in the table in Section 8.5.2). For `getXXX` methods, the driver converts the SQL type returned by the database to a Java type (using the standard mappings shown in the table in Section 8.5.1) before returning it to the `getXXX` method.

The method `registerOutParameter` always takes an SQL type as an argument, and the method `setObject` may take an SQL type as an argument.

Note that if an SQL type is supplied in its optional third argument, the method `setObject` will cause an explicit conversion of the parameter value from a Java type to the SQL type specified. If no target Sql type is supplied to `setObject`, the parameter value will be converted to the SQL type that is the standard mapping from the Java type (as shown in Section 8.5.2). The driver will perform the explicit or implicit conversion before sending the parameter to the database.

8.5 Tables for Data Type Mapping

This section contains the following tables relating to SQL and Java data types:

Section 8.5.1—SQL Types Mapped to Java Types

Section 8.5.2—Java Types Mapped to SQL Types

Section 8.5.3—SQL Types Mapped to Java `Object` Types

Section 8.5.4—Java `Object` Types Mapped to SQL Types

Section 8.5.5— Conversions by `setObject`

Section 8.5.6—SQL Types Retrieved by `ResultSet.getXXX` methods

8.5.1 SQL Types Mapped to Java Types

SQL type	Java type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
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

8.5.2 Java Types Mapped to SQL Types

This table shows the reverse mapping of Table 8.5.1, from Java types to SQL types.

Java Type	SQL type
`String`	`VARCHAR or LONGVARCHAR`
`java.math.BigDecimal`	`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`

The mapping for String will normally be `VARCHAR` but will turn into `LONGVARCHAR` if the given value exceeds the driver's limit on `VARCHAR` values. The same is true for `byte[]` and `VARBINARY` and `LONGVARBINARY` values.

8.5.3 SQL Types Mapped to Java Object Types

Since the Java built-in types such as `boolean` and `int` are not subtypes of `Object`, there is a slightly different mapping from SQL types to Java object types for the `getObject`/`setObject` methods. This mapping is shown in the following table:

SQL Type	Java Object Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
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

8.5.4 Java Object Types Mapped to SQL Types

Java Object Type	SQL Type
`String`	`VARCHAR or LONGVARCHAR`
`java.math.BigDecimal`	`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`

Note that the mapping for `String` will normaly be `VARCHAR` but will turn into `LONGVARCHAR` if the given value exceeds the driver's limit on `VARCHAR` values. The case is similar for `byte[]` and `VARBINARY` and `LONGVARBINARY` values.

8.5.5 Conversions by `setObject`

The method `setObject` converts Java object types to SQL types.

	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.math.BigDecimal	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

Conversion from Java object types to SQL types.

8.5.6 SQL Types Retrieved by `ResultSet.getXXX` Methods

An “x” means that the method *can* retrieve the SQL type. An “X” means that the method is *recommended* for the SQL type.

	TINYINT	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						
getBigDecimal	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**			
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

9
Sample Code

```
    // The following code can be used as a template.  Simply
    // substitute the appropriate url, login, and password, and then substitute the
    // SQL statement you want to send to the database.



    //----------------------------------------------------------------------------
    //
    // Module:SimpleSelect.java
    //
    // Description:Test program for ODBC API interface.  This java application
    // will connect to a JDBC driver, issue a select statement
    // and display all result columns and rows
    //
    // Product:JDBC to ODBC Bridge
    //
    // Author:Karl Moss
    //
    // Date:February, 1996
    //
    // Copyright:1990-1996 INTERSOLV, Inc.
    // This software contains confidential and proprietary
    // information of INTERSOLV, Inc.
    //----------------------------------------------------------------------------

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

class SimpleSelect {

    public static void main (String args[]) {
        String url   = "jdbc:odbc:my-dsn";
        String query = "SELECT * FROM emp";
```

```
try {

    // Load the jdbc-odbc bridge driver

    Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

    DriverManager.setLogStream(System.out);

    // Attempt to connect to a driver.  Each one
    // of the registered drivers will be loaded until
    // one is found that can process this URL

    Connection con = DriverManager.getConnection (
        url, "my-user", "my-passwd");

    // If we were unable to connect, an exception
    // would have been thrown.  So, if we get here,
    // we are successfully connected to the URL

    // Check for, and display and warnings generated
    // by the connect.

    checkForWarning (con.getWarnings ());

    // Get the DatabaseMetaData object and display
    // some information about the connection

    DatabaseMetaData dma = con.getMetaData ();

    System.out.println("\nConnected to " + dma.getURL());
    System.out.println("Driver       " +
        dma.getDriverName());
    System.out.println("Version      " +
        dma.getDriverVersion());
    System.out.println("");

    // Create a Statement object so we can submit
    // SQL statements to the driver

    Statement stmt = con.createStatement ();

    // Submit a query, creating a ResultSet object

    ResultSet rs = stmt.executeQuery (query);
```

```
            // Display all columns and rows from the result set

            dispResultSet (rs);

            // Close the result set

            rs.close();

            // Close the statement

            stmt.close();

            // Close the connection

            con.close();
        }
        catch (SQLException ex) {

            // A SQLException was generated.  Catch it and
            // display the error information.  Note that there
            // could be multiple error objects chained
            // together

        System.out.println ("\n*** SQLException caught ***\n");

        while (ex != null) {
            System.out.println ("SQLState: " +
                    ex.getSQLState ());
            System.out.println ("Message:  " + ex.getMessage ());
            System.out.println ("Vendor:   " +
                    ex.getErrorCode ());
            ex = ex.getNextException ();
            System.out.println ("");
            }
        }
        catch (java.lang.Exception ex) {

            // Got some other type of exception.  Dump it.

            ex.printStackTrace ();
        }
    }

    //-------------------------------------------------------------------
    // checkForWarning
    // Checks for and displays warnings.  Returns true if a warning
```

```
// existed
//--------------------------------------------------------------------

private static boolean checkForWarning (SQLWarning warn)
                                           throws SQLException  {
    boolean rc = false;

    // If a SQLWarning object was given, display the
    // warning messages.  Note that there could be
    // multiple warnings chained together

    if (warn != null) {
        System.out.println ("\n *** Warning ***\n");
        rc = true;
        while (warn != null) {
            System.out.println ("SQLState: " +
                warn.getSQLState ());
            System.out.println ("Message:  " +
                warn.getMessage ());
            System.out.println ("Vendor:   " +
                warn.getErrorCode ());
            System.out.println ("");
            warn = warn.getNextWarning ();
        }
    }
    return rc;
}

//--------------------------------------------------------------------
// dispResultSet
// Displays all columns and rows in the given result set
//--------------------------------------------------------------------

private static void dispResultSet (ResultSet rs)
    throws SQLException
{
    int i;

    // Get the ResultSetMetaData.  This will be used for
    // the column headings

    ResultSetMetaData rsmd = rs.getMetaData ();

    // Get the number of columns in the result set

    int numCols = rsmd.getColumnCount ();
```

```
        // Display column headings

        for (i=1; i<=numCols; i++) {
            if (i > 1) System.out.print(",");
            System.out.print(rsmd.getColumnLabel(i));
        }
        System.out.println("");

        // Display data, fetching until end of the result set

        boolean more = rs.next ();
        while (more) {

            // Loop through each column, getting the
            // column data and displaying

            for (i=1; i<=numCols; i++) {
                if (i > 1) System.out.print(",");
                System.out.print(rs.getString(i));
            }
            System.out.println("");

            // Fetch the next result set row

            more = rs.next ();
        }
    }
}
```

10 JDBC-ODBC Release Notes

10.1 JDBC-ODBC Bridge

If possible, use a Pure Java JDBC driver instead of the Bridge and an ODBC driver. This completely eliminates the client configuration required by ODBC. It also eliminates the potential that the Java VM could be corrupted by an error in the native code brought in by the Bridge (that is, the Bridge native library, the ODBC driver manager library, the ODBC driver library, and the database client library).

10.1.1 What Is the JDBC-ODBC Bridge?

The JDBC-ODBC Bridge is a JDBC driver which implements JDBC operations by translating them into ODBC operations. To ODBC it appears as a normal application program. The Bridge implements JDBC for any database for which an ODBC driver is available. The Bridge is implemented as the `sun.jdbc.odbc` Java package and contains a native library used to access ODBC. The Bridge is a joint development of Intersolv and JavaSoft.

10.1.2 What Version of ODBC Is Supported?

The bridge supports ODBC 2.x. This is the version that most ODBC drivers currently support. It will also likely work with most forthcoming ODBC 3.x drivers; however, this has not been tested.

10.1.3 The Bridge Implementation

The Bridge is implemented in Java and uses Java native methods to call ODBC.

10.1.4 Installation

The Bridge is installed automatically with the JDK as package `sun.jdbc.odbc`. See your ODBC driver vendor for information on installing and configuring ODBC. No special configuration is required for the Bridge. See your database vendor for client installation and configuration information. On Solaris, some ODBC driver managers name their libs `libodbcinst.so` and `libodbc.so`. The Bridge expects these libraries to be named `libodbcinst.so.1` and `libodbc.so.1`, so symbolic links for these names must be created.

10.2 Using the Bridge

The Bridge is used by opening a JDBC connection using a URL with the `odbc` subprotocol. See below for URL examples. Before a connection can be established, the bridge driver class, `sun.jdbc.odbc.JdbcOdbcDriver`, must either be added to the `java.lang.System` property named `jdbc.drivers`, or it must be explicitly loaded using the Java class loader. Explicit loading is done with the following line of code:

```
Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);
```

When loaded, the ODBC driver (like all good JDBC drivers) creates an instance of itself and registers this with the JDBC driver manager.

10.2.1 Using the Bridge from an Applet

JDBC used with a Pure Java JDBC driver works well with applets. The Bridge driver does not work well with applets.

10.2.2 Most Browsers Do Not Support the Bridge

Since the Bridge is an optional component of the JDK, it may not be provided by a browser. Even if it is provided, only trusted applets (those allowed to write to files) will be able to use the Bridge. This is required in order to preserve the security of the applet sandbox. Finally, even if the applet is trusted, ODBC and the DBMS client library must be configured on each client.

10.2.3 Tested Configurations

From Solaris, we have used the Bridge to access Oracle 7.1.6 and Sybase Version 10 running on Solaris. From NT, we have used the Bridge to access SQL Server 6.x.

10.2.4 ODBC Drivers Known to Work with the Bridge

Visigenic provides ODBC drivers which have been tested with the the Bridge. Drivers are available for Oracle, Sybase, Informix, Microsoft SQL Server, and Ingres. To purchase the ODBC DriverSet 2.0, please contact Visigenic sales at 415-312-7197, or visit the web site `www.visigenic.com`. The INTERSOLV ODBC driver suite should be completely compatible with the JDBC-ODBC Bridge. The following drivers have successfully passed a minimal test suite: Oracle, xBASE, Sybase (Windows NT/95 only), Microsoft SQL-Server, and Informix. To evaluate or purchase INTERSOLV ODBC drivers, please contact INTERSOLV DataDirect Sales at 1-800-547-4000 Option 2 or via the World Wide Web at `http:\\www.intersolv.com`. The MS SQL Server driver has also been used successfully on NT. Many other ODBC drivers will likely work.

10.2.5 ODBC Driver Incompatibilities

On Solaris, we have found that the Sybase ctlib-based drivers don't work because ctlib has a signal-handling conflict with the Java VM. This is likely not a problem on NT due to differences in the NT Java VM; however, this has not been verified. Some ODBC drivers only allow a single result set to be active per connection.

10.2.6 What Is the JDBC URL Supported by the Bridge?

The Bridge driver uses the `odbc` subprotocol. URLs for this subprotocol are of the form:

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

For example:

```
jdbc:odbc:sybase
jdbc:odbc:mydb;UID=me;PWD=secret
jdbc:odbc:ora123;Cachesize=300
```

10.2.7 Debugging

The Bridge provides extensive tracing when `DriverManager` tracing is enabled. The following line of code enables tracing and sends it to standard out:

```
java.sql.DriverManager.setLogStream(java.lang.System.out);
```

10.3 General Notes

The Bridge assumes that ODBC drivers are not reentrant. This means the Bridge must synchronize access to these drivers. The result is that the Bridge provides limited concurrency. This is a limitation of the Bridge. Most Pure Java JDBC drivers provide the expected level of concurrent access.