

Agave JDBC NetServer Developer's Guide (Version 1.0)

© **Agave Software Design**[™]
1997

© 1997 by Agave Software Design (Agave). All rights reserved.
Printed in the United States of America, 1997.

This manual is licensed by Agave to the user for internal use only and is protected by copyright. The user is authorized to download and print a copy of this manual if the user has purchased one or more of the Agave products described herein. All copies of this manual shall include the copyright notice contained herein. No part of this manual may be incorporated into user's documentation without prior written approval of:

Agave Software Design
701 Plano Parkway, Suite 305
Plano, Texas 75074

Phone: (972) 424-6662
Fax: (972) 424-1644

Web address: <http://www.agave.com>
Email address: info@agave.com

Windows[®] and Windows NT[®] are registered trademarks of Microsoft, Inc.
SunOS[™], Solaris[®], JAVA[™], and JDBC[™] are trademarks of Sun Microsystems, Inc.
ORACLE[®] is a registered trademark of Oracle Corporation.
UNIX[®] is a registered trademark of UNIX Systems Laboratories Inc.
HP-UX[®] is a registered trademark of Hewlett-Packard.
INTERSOLV[®] is a registered trademark of INTERSOLV, Inc.
Sybase[®] is a registered trademark of Sybase, Inc.
Adobe[®] and Adobe Acrobat[®] are registered trademarks of Adobe Systems, Inc.

Table of Contents

Introduction.....	6
Overview.....	6
About This Guide.....	7
Resources.....	7
About JDBC NetServer.....	9
Overview.....	9
What is the JDBC NetServer?.....	9
JDBC NetServer Applications.....	10
Server Application.....	11
Client API.....	12
JDBC NetBridge Application.....	13
What is JDBC?.....	14
What Is CORBA?.....	15
JDBC NetServer Data Flow.....	16
Requests to the Database.....	17
Return of Data to the Client.....	18
Building JDBC NetServer Programs.....	19
Overview.....	19
Handling Database Connections.....	21
Opening a connection.....	21
Structuring URLs.....	22
Passing Parameters and Receiving Results.....	23
Mapping SQL Data Types into Java.....	23
Complying with JDBC Standards for Asynchrony, Threading, and Transactions.....	23

Providing Cursor Support.....	23
Supporting SQL-2 Transitional Level Extensions.....	24
Supporting Permitted Variants and Extensions.....	24
Supporting Dynamic Database Access.....	24
JDBC Interface Definitions.....	25
JDBC NetServer Source Code Overview.....	26
Overview.....	26
Agave Implementation Modifications to package java.sql.....	28
Agave JDBC API Classes and Exceptions.....	29
Agave JDBC API Classes/Methods that Differ from Sun JDBC API Classes/Methods.....	30
Class agave.sql.AgaveJdbcDriver.....	31
Class agave.sql.AgaveJdbcPreparedStatement.....	32
Class agave.sql.AgaveJdbcResultSet.....	33
Class agave.sql.AgaveJdbcStatement.....	34
Example JDBC NetServer Programs.....	35
Overview.....	35
JDBC NetServer Source Code Basics.....	36
JDBC NetServer Application Demo.....	37
jdbcdemo Application User Interface.....	38
Importing Java Packages.....	38
Creating jdbcdemo Class and Class Variables	39
Creating the Application User Interface.....	40
Processing the User Action.....	43
Running the Program	44
Catching Exceptions Resulting from Query Execution.....	48
Closing the Query and Ending the Run Method.....	49
displayWarning Method Definition.....	51
displayResultSet Method Definition	53
Print and Println Method Definitions	55
main Method Definition	56
Compiling jdbcdemo.....	56
Creating a JDBC NetServer Applet.....	57
Running NetServer or NetBridge on Web Server Machine	58

Building an Applet from an Application	58
Determining Whether to Archive Applet Classes	62
Distributing the Applet and Supporting Files on the Web server	64
NetServer appletdemo Script	65
jdbcdemo Application User Interface	66
Importing Java Packages.....	68
Creating appletdemo Class, Class Variables, and User Interface.....	68
Processing the User Action.....	73
Running the Program and Ending the Run Method.....	73
Method Definitions.....	77

Introduction

Overview

Welcome to the 1.0 version of the Agave JDBC NetServer! The Agave JDBC NetServer enables developers to build Java™ programs that access various types of database systems (including Oracle, Informix, Sybase, and ODBC) across the Internet. The Agave JDBC NetServer functions as the interface between a Java program running on a Web browser and a remote database on the Internet or Intranet.

Thanks to its implementation of the standards established by CORBA (Common Object Request Broker Architecture) technology and Sun Microsystems' JDBC 1.1 API, the Agave JDBC NetServer provides an open solution to database access. It enables developers to concentrate on building programs, without also having to build specialized back-end tools to pass data between the Web client and the remote database server.

About This Guide

The *JDBC NetServer Developer's Guide* is designed for software developers who are developing Java programs to access databases across the Internet or Intranet using JDBC NetServer as an interface. This guide provides an overview of how the JDBC NetServer applications work together to pass data between a Web Client and a remote database. It also provides information about using JDBC code to create JDBC Java applications and applets that implement the JDBC NetServer Client API.

Resources

This *JDBC NetServer Developer's Guide* assumes that you have some knowledge of SQL, Java, and Sun's JavaSoft JDBC™. To learn more about these languages, see the following sources.

Recommended World Wide Web sites:

- For information on SQL, see <http://www.inquiry.com/techtips/thesqlpro>
- For information on Java, see <http://www.javasoft.com>
- For information on Sun's Java Development Kit 1.1 (JDK 1.1), see <http://www.javasoft.com:80/products/jdk/1.1/docs/index.html>
- To view and/or download the JDBC 1.1 Specification or the JDBC 1.1 API documentation, see <http://splash.javasoft.com/jdbc/>
- For general reference information about using JDBC, see <http://www.javasoft.com:80/products/jdk/1.1/docs/guide/jdbc/index.html>

- For information on various object standards and technologies, see the Object Management Group's home page at <http://www.omg.org/>

Recommended books:

- For information on SQL, see *Instant SQL Programming*, by Joe Celko.
- For information on Java, see:
 - *Core Java* by Gary Cornell and Cay S. Horstmann
 - *Teach Yourself JAVA in 21 Days, Professional Reference Edition*, by Laura Lemay, Charles L. Perkins, and Michael Morrison

About JDBC NetServer

Overview

This chapter provides a general overview of the JDBC NetServer application. It provides information that answers the following questions:

- [What is the JDBC NetServer?](#)
- [What is JDBC?](#)
- [What Is CORBA?](#)
- [How does the JDBC NetServer handle client-server data flow?](#)

What is the JDBC NetServer?

The Agave JDBC NetServer enables programmers to build Java applications that access various types of database systems (including Oracle, Informix, Sybase, and ODBC) across the Internet. JDBC NetServer provides the interface between a Java JDBC applet or application running on a Web browser and the database being accessed across the Internet by the Java program.

JDBC NetServer implements the client-server data-exchange standards established by the Object Management Group's (OMG) CORBA and Sun Microsystem's JDBC API. Therefore, the JDBC NetServer provides true interoperability between a Web user's machine and a remote database machine. With JDBC NetServer, a Java program running on a browser simply needs to incorporate JDBC statements that direct database access requests to use JDBC NetServer as a bridge to the database.

JDBC NetServer Applications

The JDBC NetServer package includes three applications:

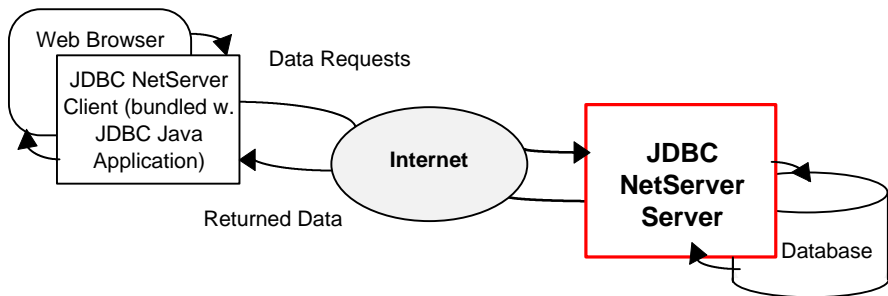
- [*Server application*](#)—to be installed on your database server
- [*Client API*](#)—to be installed on your JDBC Java application development machine
- [*JDBC NetBridge*](#)—to be installed on your Web server, if needed for your configuration

You may need to install these applications on one machine or on two or three separate machines, depending on your environment.

Server Application

The JDBC NetServer Server application must be installed on the same machine as your database. The NetServer Server enables the database server to pass Java-based data requests sent across the Internet (via the JDBC NetServer Client API) to the local database. It also enables the database server to return data retrieved from the database across the Internet to the requesting Java application (which receives the data via the Client API).

The following diagram illustrates the role of the Server application in transferring data between a Web-based JDBC client application and a database:

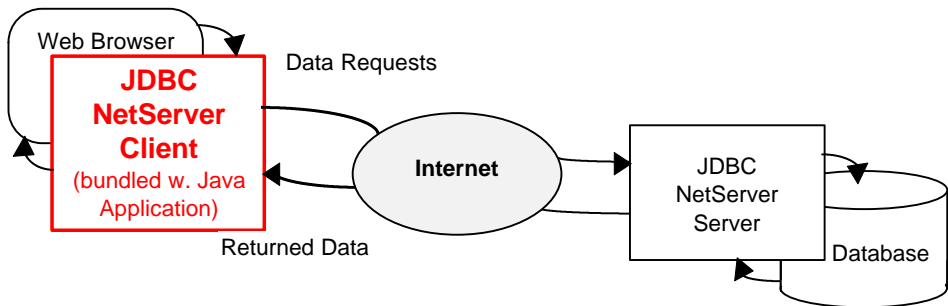


Client API

The JDBC NetServer Client API must be installed on the machine used to develop JDBC Web applications and applets. (This may be your Web server machine or another machine.) The Client API is part of a development package that enables JDBC-based client programs to access a database across the Internet. Once the Client API is installed on the development machine, it is available to be bundled with JDBC applications and applets for distribution. (The finished applications or applets may be installed on the Web server or distributed manually.)

When bundled with a JDBC-based Java client application or applet, the Client API sends data requests from the client across the Internet. The requests are received by the NetServer Server package on the database machine and passed on to the database. When data is returned across the Internet from the database (via the Server), the Client API then receives the data and passes it to the client.

The following diagram illustrates the role of the Client API in transferring data between a Web-based JDBC client application and a database:



JDBC NetBridge Application

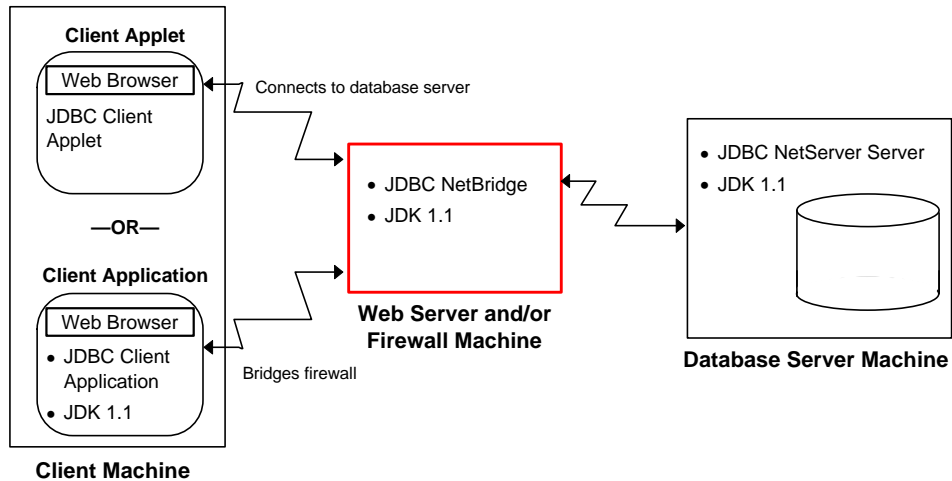
If needed for your environment, the JDBC NetBridge application must be installed on your Web server. NetBridge enables applets to connect to the database when the Web server and the database server are on different machines.

Why do applets require NetBridge in this case? Client applets must be installed on and downloaded from the Web server. And, as stated in the [Server Application](#) section, the Server application must be installed on the database server. **Applets can connect (via the network) only to the server from which they were downloaded.**

If the Web server and database server are on different machines, you can install NetBridge on the Web server to bridge the gap between the client and the database server. When NetBridge is running on the Web server, it is connected to active client applets on the Web server and to the NetServer Server application on the database server. NetBridge then functions for client applets as a proxy NetServer Server.

That is, to client applets, NetBridge appears to be the NetServer Server, so NetBridge receives the applets' data requests. Then NetBridge forwards the requests to the NetServer Server for processing. NetBridge likewise forwards returned data from the Server application to client applets.

The following diagram illustrates how data requests and returned data travel through the NetBridge between a Web-based JDBC client applet or application and a database:



What is JDBC?

JDBC (Java Database Connectivity) was developed by Sun Microsystems' JavaSoft division as a Java Application Programming Interface (API) to SQL databases. The JDBC API enables developers to write code that is independent of the specific DBMS or database connectivity mechanism being used. It is a generic SQL database access framework that provides a standard interface on top of various types of database connectivity modules.

Sun has based the JDBC API on the X/Open SQL Call Level Interface (CLI), which is also the basis for Microsoft's ODBC (Open Database Connectivity) interface. Calls travel from Java to C code through the JDBC interface, and thus bypass the problems that occur with direct calls from Java to C in areas such as security, robustness, and portability.

In and of itself, Sun's JDBC API promises to be an invaluable tool for exchanging data between Java programs and SQL databases. However, to access remote databases on the Internet, an additional layer of functionality is required. Alone, JDBC cannot send or receive data across the Internet. It does not provide methods for converting data requests and returned data into structures that are transmissible over the Internet.

Agave has solved these limitations by extending the JDBC API over the Internet. It provides the functionality needed to transmit data requests and data between Java-based Web applications and SQL-based remote databases, using industry-standard CORBA networking tools and protocols.

What Is CORBA?

The data conversion and transmission methods incorporated by the JDBC NetServer make it an open solution for Internet database access needs. The use of CORBA technology enables client and server applications to receive object requests and returned data objects in whole, regardless of differences in machine platforms, operating systems, and programming languages.

What is CORBA? CORBA (Common Object Request Broker Architecture) is a standard for distributed objects. CORBA was developed by the Object Management Group (OMG), a consortium dedicated to promoting the theory and practice of object technology for the development of distributed computing systems.

CORBA provides the mechanisms that enable objects to transparently make requests and receive responses, as defined by OMG's ORB (Object Request Broker). The CORBA ORB is an application framework that provides interoperability between objects, regardless of the languages they are built in or the machines they are running on, in heterogeneous distributed environments.

CORBA defines the OMG Interface Definition Language (IDL) and the APIs that enable client/server object interaction within an ORB implementation.

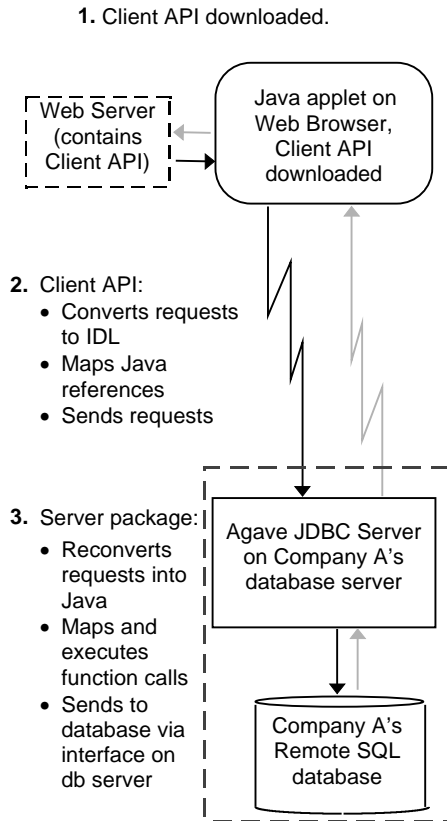
The ORB is the middleware that forms the client-server relationships between objects. The OMG IDL expresses the interface between a sending object and target object, and defines the message format in an interface known to the system, enabling the system's communications infrastructure to handle the details. The IDL makes the interfaces accessible to objects written in virtually any programming language. JDBC NetServer incorporates Java IDL, which is based on the industry-standard OMG IDL.

A client using an ORB can transparently invoke a method on a server object, on the same machine or across a network. The ORB intercepts the call, and finds an object that can implement the call's request, pass its parameters, invoke its method, and return the results. The client does not have to know the object's location, programming language, operating system, or other system aspects that are not part of an object's interface.

JDBC NetServer Data Flow

Following are examples of the data flow between a Java Client applet and a database across the Internet using the JDBC NetServer Client API and Server .

Requests to the Database



1. When an end-user on a Web browser clicks on a link on Company A's Web page, a Java JDBC applet downloads and executes the Client API.

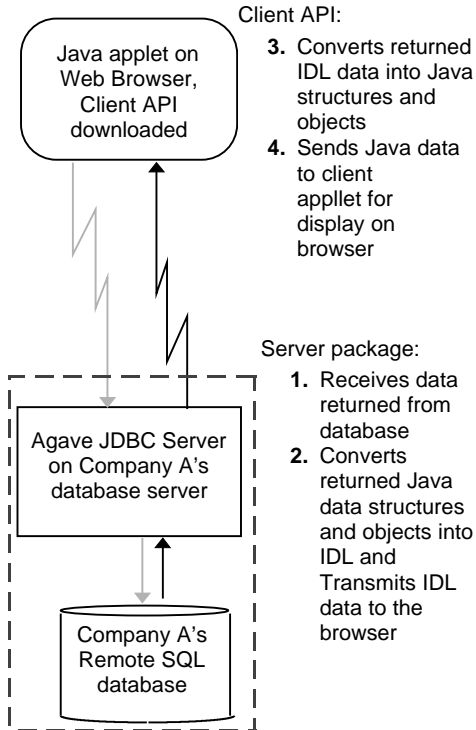
2. The Client API converts JDBC requests into Interface Definition Language (IDL) and maps Java references to the object on the database server. Then the Client API sends the data requests across the Internet to the remote database server.

3. After the data requests arrive at Company A's database server, the Server package converts transmitted data back into Java from IDL.

The JDBC Server package then maps and executes function calls from the Java client applet. It maps the function calls to corresponding functions in Sun's JDBC API on the database server.

4. The database server further processes the data request and receives the results to return to Server package.

Return of Data to the Client



Client API:

3. Converts returned IDL data into Java structures and objects
4. Sends Java data to client applet for display on browser

Server package:

1. Receives data returned from database
2. Converts returned Java data structures and objects into IDL and Transmits IDL data to the browser

1. After Company A's database has processed requests it received via the JDBC NetServer Server, the Server receives data returned from database.
2. The Server converts returned Java data structures and objects into IDL, and then transmits the IDL data across the Internet to the Client on the Web browser.
3. On the browser, the JDBC Client API converts returned IDL data into Java structures and objects, which it then returns to the Java client applet.
4. The end-user views the returned data and makes a tremendous, life-changing decision regarding Company A!

And the end-user spends only a matter of seconds waiting on requested information to be returned!

The entire data request and return process can occur in less than one second. However, keep in mind that the actual performance is limited by database performance and network bandwidth.

Building JDBC NetServer Programs

Overview

Agave built the JDBC NetServer in accordance with the Sun JavaSoft JDBC 1.1 Specification. With a few modifications, the procedures and source code required to build JDBC programs for JDBC NetServer are the same as those required to build JDBC programs using the standard Sun JavaSoft JDBC 1.1 API.

JDBC NetServer programs use classes in the Agave JDBC API for SQL functionality. The Agave JDBC API contains all classes that are implemented or derived uniquely by JDBC NetServer from the interfaces and classes in the JavaSoft JDBC 1.1 API.

The JavaSoft JDBC 1.1 API is included in Sun's Java Development Kit (JDK) 1.1. Database server, client development, and Web server machines require Sun's JDK 1.1 to run JDBC NetServer components. JDK 1.1 is part of the Agave JDBC NetServer CD-ROM installation package.

This chapter outlines the steps described in the JDBC 1.10 Specification and explains some key differences between building programs for the Agave JDBC API and the JavaSoft JDBC API.

When designing your JDBC program for the Agave JDBC NetServer, you must make sure that your program can:

1. [Connect to databases](#)
2. [Pass parameters and receive results](#)
3. [Map SQL data types into Java](#)
4. [Comply with JDBC standards concerning asynchrony, threading, and transaction execution](#)
5. [Provide appropriate cursor support](#)
6. [Support selective SQL-2 transitional level extensions](#)
7. [Support permitted variants and vendor-specific extensions, as necessary, for compatibility with your database system](#)
8. [Support dynamic database access for metadata and dynamically typed data access](#)

This documentation of Agave's JDBC NetServer does not include detailed information about the procedures for building JDBC source code. You can find detailed information in Sun's JavaSoft JDBC 1.10 Specification by downloading the JDBC 1.10 Specification as a postscript file or Adobe Acrobat Portable document format (PDF) file from the following Web site:

<http://splash.javasoft.com/jdbc/>

You can also access documentation of the JDBC 1.10 API package (java.sql).

You can find detailed information about the JDK 1.1 at <http://www.javasoft.com:80/products/jdk/1.1/docs/index.html>

You can find additional JDBC reference information at <http://www.javasoft.com:80/products/jdk/1.1/docs/guide/jdbc/index.html>

We recommend that you download and print the JDBC Specification to use as a guideline for developing your JDBC program for the Agave JDBC NetServer. Then modify the procedures described in the JDBC Specification according to the information provided in the following topics of this chapter. Also, use the sample source code provided with the JDBC NetServer package and examined in the [Example JDBC NetServer Programs](#) chapter of this guide to help you get started with your JDBC program development.

Handling Database Connections

In general, the procedures for handling connections using the JDBC NetServer are the same as those outlined in the JDBC 1.1 Specification.

However, you need to specify the `AgaveJdbcDriver` when you make the SQL database connection. The `AgaveJdbcDriver` routes `java.sql` interface calls to the `agave.sql` classes that implement the interfaces. The program source code must load the `AgaveJdbcDriver`, and the `AgaveJdbcDriver` object must register itself with the `DriverManager` (as described in the JavaSoft JDBC 1.1 specification).

Opening a connection

The Agave JDBC API handles the opening of a database connection in almost the same way as the JavaSoft JDBC API. However, the URL syntax in connection statements must be structured to include the JDBC NetServer address, as explained in the following topic, [Structuring URLs](#).

Structuring URLs

You must modify the URL syntax to access a database through the JDBC NetServer.

The URL syntax for JDBC programs using the JDBC NetServer must be structured to include the network address of the JDBC NetServer in the JDBC URL.

URL Syntax

Whereas basic JDBC URLs would be structured as:

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

(where `subprotocol` names the database connectivity mechanism, and `subname` names the data source).

JDBC URLs that travel through the Agave JDBC NetServer should be structured (with no spaces between characters) as:

```
<domainname>:<port#>/JDBCNetServer/jdbc:  
<subprotocol>:<subname>
```

Example URL

An example URL to access a database through the JDBC NetServer is (with no spaces between characters):

```
mezcal.agave.com:4493/JDBCNetServer/jdbc:ora7
```

Passing Parameters and Receiving Results

The Agave JDBC API handles the passing of parameters and receipt of results in the same way as the JavaSoft JDBC API in terms of handling query results and data truncation.

You do not need to modify the procedures described in the JavaSoft JDBC 1.1 Specification to pass parameters and receive results.

Mapping SQL Data Types into Java

The Agave JDBC API handles the mapping of SQL data types into Java in the same way as the JavaSoft JDBC API.

You do not need to modify the procedures described in the JavaSoft JDBC 1.1 Specification to do this mapping.

Complying with JDBC Standards for Asynchrony, Threading, and Transactions

The Agave JDBC API complies fully with the standards set forth in the JavaSoft JDBC 1.1 Specification concerning asynchrony, threading, and transaction execution.

Providing Cursor Support

The Agave JDBC API handles the provision of cursor support in the same way as the JavaSoft JDBC API.

You do not need to modify the procedures described in the JavaSoft JDBC 1.1 Specification to provide cursor support.

Supporting SQL-2 Transitional Level Extensions

The Agave JDBC API supports selective SQL-2 Transitional Level extensions in the same way as the JavaSoft JDBC API.

You do not need to modify the procedures described in the JavaSoft JDBC 1.1 Specification to support these extensions.

Supporting Permitted Variants and Extensions

The Agave JDBC API supports permitted variants and vendor-specific extensions the same way as the JavaSoft JDBC API.

You do not need to modify the procedures described in the JavaSoft JDBC 1.1 Specification to support permitted variants and extensions.

Supporting Dynamic Database Access

The Agave JDBC API handles dynamic database access in the same way as the JavaSoft JDBC API in terms of accessing metadata. Likewise, the Agave JDBC API handles dynamically typed data access as described in the JavaSoft JDBC 1.1 Specification

JDBC Interface Definitions

The NetServer-specific Agave JDBC API implements all JDBC core java.sql interfaces and classes listed in the JavaSoft JDBC 1.1 Specification.

The Agave JDBC API includes classes that are implemented uniquely by NetServer from JavaSoft JDBC interfaces (or, in the case of the SQLException and SqlWarning classes, derived uniquely from JavaSoft JDBC classes).

The Agave JDBC classes are included in the agave.sql package. JDBC NetServer implements interfaces as classes with the prefix *AgaveJdbc*. The AgaveJdbc driver forwards program calls to java.sql interface methods to the appropriate agave.sql classes that implement the java.sql interfaces.

The [JDBC NetServer Source Code Overview](#) chapter provides detailed information about implemented and derived classes in the Agave JDBC API.

JDBC NetServer Source Code Overview

Overview

The Agave JDBC NetServer package requires Sun's JDK 1.1 to run on a database server, development system, or Web server machine. The JDK 1.1 includes Java packages of classes and interfaces that are needed to create Java applications and applets. It includes the JDBC API as a standard Java package, called *java.sql*, to provide Java programs with database connectivity. (The JDK 1.1 is available on the Agave JDBC NetServer CD ROM or from Sun Microsystems' Web site, as discussed in the *Agave JDBC NetServer Installation Guide*.)

The JDBC NetServer package also includes a separate package of Java language classes that are implemented from core *java.sql* interfaces or derived from certain *java.sql* classes. These classes comprise the Agave JDBC API, in a package called *agave.sql*.

Because Agave built the JDBC NetServer according to the Sun Microsystems' JavaSoft JDBC 1.1 Specification, Agave JDBC is almost the same as Sun JDBC. The only way that some of the *Agave.sql* classes differ from their *java.sql* counterparts is that the names of *agave.sql* classes implemented from *java.sql* interfaces or derived from *java.sql* classes have an *AgaveJdbc* prefix. However, some Agave classes and methods require important modifications to enable declarations to be passed across the Internet.

The instruction in your JDBC NetServer source code to load and register the `AgaveJdbcDriver` ensures that calls made to `java.sql` interfaces or classes use `agave.sql` implementations or derivations.

This documentation of Agave's JDBC NetServer does not provide detailed information about all of the `java.sql` or `agave.sql` classes and interfaces used for building JDBC NetServer source code. Detailed documentation of the Sun JDBC API is available from various sources, in various formats, and can be used to learn about the interfaces implemented and classes extended by the `agave.sql` package.

Instead, this chapter identifies the overall differences between the Agave JDBC class implementations and the Sun JDBC interfaces and classes. Also, for each Agave JDBC class and/or method that is different in usage or functionality from Sun's JDBC classes and methods, this chapter explains the differences.

This chapter provides the following information about the Agave implementation of the JavaSoft JDBC API:

- [Agave implementation modifications to package java.sql](#)
- [Agave JDBC API classes and exceptions](#)
- [Agave JDBC API classes and/or methods that differ in functionality or usage from Sun JDBC API classes/methods](#)
 - [Class `agave.sql.AgaveJdbcDriver`](#)
 - [Class `agave.sql.AgaveJdbcPreparedStatement`](#)
 - [Class `agave.sql.AgaveJdbcResultSet`](#)
 - [Class `agave.sql.AgaveJdbcStatement`](#)

You can find detailed information about JDBC code in the online documentation distributed by Sun with the JDK 1.1. Or you can download the JDBC 1.10 JDBC API online documentation as a postscript or PDF file from the following Web site:

<http://splash.javasoft.com/jdbc/>

Use this chapter as a supplement to Sun's JDBC source definitions when you build applications that use the Agave JDBC NetServer.

Agave Implementation Modifications to package java.sql

Overall differences between the Sun JDBC API and the Agave JDBC API include:

- Core interfaces in the Sun JDBC API are implemented as *AgaveJdbc* classes in the Agave JDBC API.
- The Agave JDBC API implements abstract methods; therefore these method declarations do not contain the term *abstract*.

For example, rather than a declaration starting as: `public abstract final` it would start as: `public final`

Agave JDBC API Classes and Exceptions

The `agave.sql` package implements certain core `java.sql` interfaces as classes; it also derives classes from the `java.sql` `SQLException` and `SQLWarning` classes.

The following table identifies `agave.sql` classes that implement `java.sql` interfaces:

This <code>agave.sql</code> class	Implements this <code>java.sql</code> interface
<code>AgaveJdbcConnection</code>	<code>Connection</code>
<code>AgaveJdbcDatabaseMetaData</code>	<code>DatabaseMetaData</code>
<code>AgaveJdbcDriver</code>	<code>Driver</code>
<code>AgaveJdbcResultSet</code>	<code>ResultSet</code>
<code>AgaveJdbcResultSetMetaData</code>	<code>ResultSetMetaData</code>
<code>AgaveJdbcStatement</code>	<code>Statement</code>
<code>AgaveJdbcCallableStatement</code>	<code>CallableStatement</code>
<code>AgaveJdbcPreparedStatement</code>	<code>PreparedStatement</code>

The following table identifies `agave.sql` classes that are derived from `java.sql` classes:

This <code>agave.sql</code> class	Is derived from this <code>java.sql</code> class
<code>AgaveJdbcSQLException</code>	<code>SQLException</code>
<code>AgaveJdbcSQLWarning</code>	<code>SQLWarning</code>

Agave JDBC API Classes/Methods that Differ from Sun JDBC API Classes/Methods

Agave JDBC API classes and/or methods that differ from the Sun JDBC API implementation include (listed by class):

- [Class `agave.sql.AgaveJdbcDriver`](#)
 - `connect`
- [Class `agave.sql.AgaveJdbcPreparedStatement`](#)
 - `setAsciiStream` (not fully implemented in `agave.sql`; can be called, but does nothing)
 - `setBinaryStream` (not fully implemented in `agave.sql`; can be called, but does nothing)
 - `setUnicodeStream` (not fully implemented in `agave.sql`; can be called, but does nothing)
- [Class `agave.sql.AgaveJdbcResultSet`](#)
 - `getObject (int)`
 - `getObject (string)`
 - `clearWarnings` (can be called, but does nothing)
- [Class `agave.sql.AgaveJdbcStatement`](#)
 - `clearWarnings` (not fully implemented in `agave.sql`; can be called, but does nothing)
 - `getWarnings` (not fully implemented in `agave.sql`; can be called, but does nothing)
 - `cancel` (not fully implemented in `agave.sql`; can be called, but does nothing)

Class `agave.sql.AgaveJdbcDriver`

public class **Driver**

In `agave.sql`, only one driver is currently allowed. The driver **Jdbc-odbc** is hard-coded on the server.

connect

```
public Connection connect(String url, Properties
info) throws SQLException
```

Tries to make a database connection to the given URL. The driver should return *null* if it is the wrong type of driver to connect to the given URL. It should raise an `SQLException` if it is the right driver but cannot connect properly to the database.

The `java.util.Properties` argument can pass arbitrary string tag/value pairs as connection arguments. Normally, the `Properties` argument should include a minimum of *user* and *password* properties.

In `agave.sql`, the URL must contain the path to the server object, as in the following example (with no spaces between items):

```
mezcal.agave.com:4499/JDBCNetServer/jdbc:subprotocol:
subname
```

`mezcal.agave.com` is the Internet domain name. `4499` is the port number. `JDBCNetServer` is the server object name. `Subprotocol` is the database connectivity mechanism, and `subname` is the data source.

Parameters:

- url** URL of the database to connect to. **In *agave.sql***, this is: `jdbc:subprotocol:subname`
- info** List of arbitrary string tag/value pairs for use as connection arguments. Normally at least a *user* and *password* property should be included.

Returns:

Connection to the URL.

Class *agave.sql.AgaveJdbcPreparedStatement*

public class **PreparedStatement**

setAsciiStream

Not fully implemented in *agave.sql*; can be called, but does nothing.

setUnicodeStream

Not fully implemented in *agave.sql*; can be called, but does nothing.

setBinaryStream

Not fully implemented in *agave.sql*; can be called, but does nothing.

Class `agave.sql.AgaveJdbcResultSet`

public class **ResultSet**

getObject

```
public Object getObject(int columnIndex) throws  
SQLException
```

Gets a database column value as a Java object. **In `agave.sql`**, first gets the value of the column as a string, then casts the string into an object.

This method returns the value of the given column as a Java object, which defaults to the Java Object type corresponding to the column's SQL type, according to the mapping described in the JDBC Specification.

This method can also be used to read database-specific abstract data types.

Parameters:

`columnIndex` Indexes first column as 1, second as 2, etc.

Returns:

A `java.lang.Object` containing the column value.

getObject

```
public Object getObject(String columnName) throws  
SQLException
```

Gets a parameter value as a Java object. **In `agave.sql`**, first gets the value of a column as a string, then casts the string into an object.

This method returns the value of the given column as a Java object, which defaults to the Java Object type corresponding to the column's SQL type, according to the mapping described in the JDBC Specification.

This method can also be used to read database-specific abstract data types.

Parameters:

`columnName` SQL name of the column.

Returns:

A `java.lang.Object` containing the column value.

clearWarnings

Not fully implemented in `agave.sql`; can be called, but does nothing.

Class `agave.sql.AgaveJdbcStatement`

```
public class Statement
```

cancel

Not fully implemented in `agave.sql`; can be called, but does nothing.

clearWarnings

Not fully implemented in `agave.sql`; can be called, but does nothing.

getWarnings

Not fully implemented in `agave.sql`; can be called, but does nothing.

Example JDBC NetServer Programs

Overview

This chapter examines the source code of two example JDBC NetServer demonstration programs: the `jdbcdemo` application and the `appletdemo` applet. It analyzes the application source code in detail, provides tutorial information about creating an applet based on an existing application, and highlights the main differences between the application source code and the applet source code. This chapter provides the following information:

- [Description of JDBC NetServer source code basics](#)
- [Analysis of the NetServer application demo source](#)
- [Instructions for creating a NetServer applet from an application](#)
- [Analysis of the NetServer applet demo source](#)

JDBC NetServer Source Code Basics

If you are familiar with Java, you will probably find it quite easy to create JDBC NetServer applets and applications.

As the example programs illustrate, JDBC NetServer program script is in the Java language. Like other Java programs, NetServer programs import class packages, including `java.sql` and `agave.sql`. The `java.sql` package defines all interfaces, classes, and exceptions in Sun Microsystems' JDBC API, which is part of Sun's JDK 1.1, and which provides Java programs with database connectivity.

The `agave.sql` package defines the Agave JDBC API. The Agave JDBC API is Agave's implementation of `java.sql` elements used uniquely by JDBC NetServer client programs. It includes classes implemented or derived from certain `java.sql` interfaces and classes. As discussed in the [JDBC NetServer Source Code Overview](#) chapter, `agave.sql` classes function the same as their `java.sql` counterparts except for some modifications to achieve Internet connectivity.

NetServer programs use the `registerDriver` method of the `java.sql.DriverManager` class to register the `AgaveJdbcDriver` with the `DriverManager` so that the Agave JDBC API is used for SQL functions.

You can find the JDBC API 1.1 Specification and the JDBC API 1.1 package documentation on Sun's Web page at the following site: <http://splash.javasoft.com/jdbc/>.

You can find information about JDK 1.1 at <http://www.javasoft.com:80/products/jdk/1.1/docs/index.html>.

You can find additional JDBC reference information at <http://www.javasoft.com:80/products/jdk/1.1/docs/guide/jdbc/index.html>.

JDBC NetServer Application Demo Script

This section examines the program code for the `jdbcdemo.java` application. It illustrates the application's [user interface](#) and then examines the sections of the script, which perform the following tasks:

- [Import Java language packages to be used by the program](#)
- [Create the `jdbcdemo` class and class instance variables](#)
- [Construct the `jdbcdemo` application program and interface](#)
- [Process the selected user action](#)
- [Run methods that execute the user's query](#)
- [Catch exceptions resulting from query execution](#)
- [Close the query and the run program](#)
- [Define the `jdbcdemo` class `displayWarning` method](#)
- [Define the `jdbcdemo` class `displayResultSet` method](#)
- [Define the `jdbcdemo` class `print` and `println` methods](#)
- [Define the `jdbcdemo` class `main` method and close the `jdbcdemo` class definition](#)

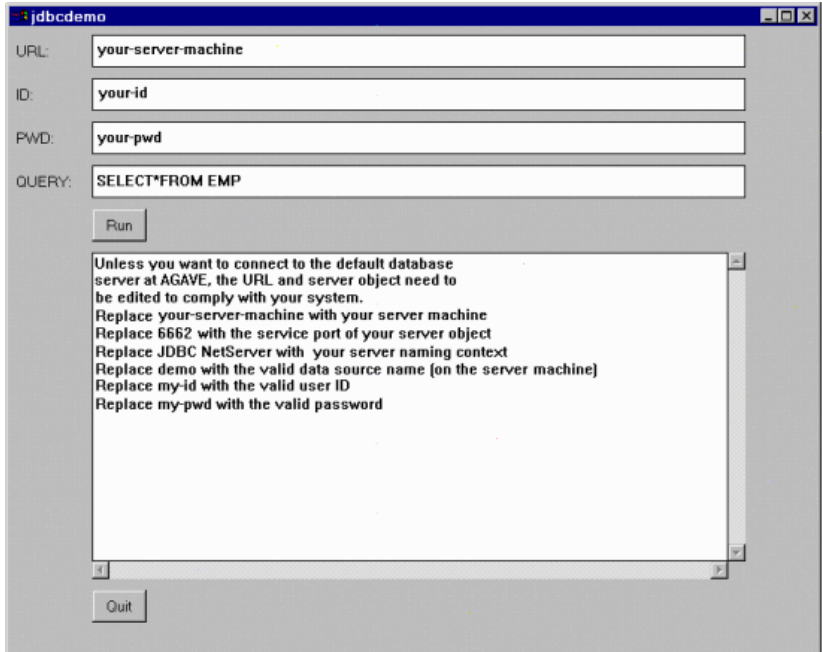
You can view this source code online by opening the `jdbcdemo.java` file in the appropriate location on your machine. The file's path depends on your machine's operating system.

For	The <code>jdbcdemo.java</code> path is:
Windows 95/NT	<code>[InstallDir]\NetServer\client\jdbcdemo.java</code>
UNIX	<code>[InstallDir]/NetServer/client/jdbcdemo.java</code>

After you make changes to the `jdbcdemo.java` file or create your own application, you must compile the source code as described in [Compiling `jdbcdemo`](#) on page 56.

jdbcdemo Application User Interface

Following is an illustration of the JDBC Demo application user interface, which enables end-users to query a database:



Importing Java Packages

The import statements in the first lines of the script import the following Java classes:

- All `agave.sql` package classes
- All `java.io` package classes
- All `java.sql` package classes (Sun's JDBC API)
- `java.net.URL` class
- All `java.awt.` package classes

The script uses classes imported from `agave.sql` to execute SQL queries, from `java.io` to manage input and output, from `java.sql` to provide SQL connection information, from `java.net` to specify a connection address, and from `java.awt` to create a user interface. (Like other Java programs, it automatically imports the `java.lang` package.)

```
import agave.sql.*
import java.io.*;
import java.sql.*;
import java.net.URL;
import java.awt.*;

//-----
//File:          jdbcdemo.java

//Description:  Test program for JDBC driver(remote database
//connection). This java application will connect to a JDBC
//driver, issue a select statement, and display all
//resulting columns and rows.

//Date:         January 6, 1997


//Programmer:   Hoang Bui
//-----
```

Creating `jdbcdemo` Class and Class Variables

The next section of the script begins the creation of the `jdbcdemo` class.

- The [first line \(labeled ①\)](#) creates the `jdbcdemo` main class and indicates that the `jdbcdemo` class extends the `Frame` class, which is part of the `java.awt` package.
- The [next lines \(②\)](#) declare instance variables for the `jdbcdemo` class, including string, button, text field, text area, and label variables to be used in other locations of the script. The first line of this block sets the driver variable to null.

```
class jdbcdemo extends Frame { — ①
    Driver driver = null;
    String url;
    String user;
    String pwd;
    String sql;
    Button runButton;
    Button quitButton;
    TextField urlField;
    TextField userField;
    TextField pwdField;
    TextField queryField;
    TextArea outArea;
    Label label;
} — ②
```



Constructing the Application User Interface

The next section of the script creates the application window. Various statements define the field label, text field, text area, and button objects that appear on the window, and that enable end users to interact with the application. (Class constructors and methods used to create these objects are included in the JDK 1.1 java.awt package.)

- The [lines labeled ①](#) begin the jdbcdemo constructor method that creates the application interface, and set the layout manager to null (thereby using the implicit BorderLayout method to create a window border).
- [②](#) creates the URL: text field label and the corresponding urlField text field object, which will have a default entry of your-server-machine:6662/JDBCNetServer/jdbc:demo.
- [③](#) creates the ID: text field label and the corresponding userField text field object, which will have a default entry of your-id.

- [④](#) creates the `PWD:` text field label and the corresponding `pwdField` text field object, which will have a default entry of `your-pwd`.
- [⑤](#) creates the `QUERY:` text field label and the corresponding `queryField` text field object, which will have a default entry of `SELECT * FROM EMP`.
- [⑥](#) creates the **Run** button.
[⑦](#) creates an output text area object called `outArea`, and prints lines of user instructions into this area by calling the `println` method. (The `println` method for the `jdbcdemo` class is created in the section of code on page 55.)
- [⑧](#) creates the **Quit** button.
- In [⑨](#), the `try` method sets the driver variable to a new `AgaveJdbcDriver` class object and calls the `registerDriver` method of the `DriverManager` class to make the driver known to the `DriverManager`. This method registers the `AgaveJdbcDriver` driver, which directs the application to the Agave JDBC API classes implemented in the `agave.sql` package for SQL calls.

Also, this block catches `SQLException` objects generated while loading or registering the driver. (The exceptions are held in the `ex` variable created by the `java.sql.SQLException` constructor).

The final line of this block ends the `jdbcdemo` constructor.

```
public jdbcdemo(String title, int width, int height) {
  ①  super(title);
     setBackground(Color.lightGray);

     setLayout(null);

     ②  label = new Label("URL:");
        add(label);
        label.reshape(10, 30, 60, 30);

        urlField = new TextField
        ("your-server-machine:6662/JDBCNetServer/jdbc:demo");
        add(urlField);
        urlField.reshape(80, 30, 600, 30);

     ③  label = new Label("ID:");
        add(label);
        label.reshape(10, 70, 60, 30);

        userField = new TextField("your-id");
        add(userField);
        userField.reshape(80, 70, 60, 30);

     ④  label = new Label("PWD:");
        add(label);
        label.reshape(10, 110, 60, 30);

        pwdField = new TextField("your-pwd");
        add(pwdField);
        pwdField.reshape(80, 110, 600, 30);

     ⑤  label = new Label("QUERY:");
        add(label);
        label.reshape(10, 150, 60, 30);

        queryField = new TextField("SELECT * FROM EMP");
        add(queryField);
        queryField.reshape(80, 150, 600, 30);

     ⑥  runButton = new Button("Run");
        add(runButton);
        runButton.reshape(80, 190, 50, 30);
}
```



```

⑦ [
    outArea = new TextArea("", 5 60);
    add(outArea);
    outArea.reshape(80, 230, 600, 300);
    println("Unless you want to connect to the default
            database");
    println("server at AGAVE, the url and server object need");
    println("to be edited to comply with your system.");
    println("Replace your-server-machine with your server
            machine.");
    println("Replace 6662 with the service port of your server
            object.");
    println("Replace JDBCNetServer with your server naming
            context.");
    println("Replace demo with the valid data source name(on
            the server machine).");
    println("Replace your-id with the valid user ID.");
    println("Replace your-pwd with the valid password.");
]

⑧ [
    quitButton = new Button("Quit");
    add(quitButton);
    quitButton.reshape(80, 540, 50, 30);
]

⑨ [
    try {
        driver = newAgaveJdbcDriver();
        DriverManager.registerDriver(driver) ;
    }
    catch (SQLException ex) { }
]

```




Processing the User Action

The next section of the script creates an action method in the `jdbcdemo` class that determines what action the program takes when the user clicks the **Quit** button or the **Run** button.

- As shown in [①](#), if the user clicks **Quit**, the user will exit the applet.
- As shown in [②](#), if the user clicks **Run**, the program will execute the `run` method, which is defined in the script on page 47.

```
public boolean action(Event e, Object arg)
{
  ① if(e.target instanceof Button) {
    if("Quit".equals(arg))
    {
      this.dispose();
      System.exit(0);
    }
  }
  ② if("Run".equals(arg))
  {
    run();
  }
  return false;
}
```



Running the Program

The next section of the script begins running the program, as defined by the `run` method, when the user clicks the **Run** button. The code in this section creates the parts of the `run` method that connect to the database, execute the query, and display the query results.

- ① declares the `run` method of the `jdbcdemo` class, and gets values for the `url`, `query`, `user`, and `pwd` variables that were created under the `jdbcdemo` class declaration. The program calls the `getText` method of the `java.awt.TextComponent` class for the `urlField`, `queryField`, `userField`, and `pwdField` objects to get values entered in these text fields.
- ② declares the connection variable of the `java.sql.Connection` class and the statement variable of the `java.sql.Statement` class and sets them as null.

- [③](#) provides commented instructions to try logging SQL function calls into the `jdbcdemo.log` file and to handle resulting IO exceptions (using the `setLogStream` method of the `java.sql.DriverManager` class, as well as `java.io` package functions).
- In [④](#), `try` begins a block of code that connects to the database and executes a query.
- [⑤](#) gets a connection to the database (and creates the connection object) by calling the `getConnection` method in the `java.sql.DriverManager` class, using the `url`, `user`, and `pwd` variable values defined in [②](#) as parameters. The `getConnection` method selects the jdbc driver and establishes the connection to the database.

These lines also include methods to display and clear warnings if problems occur while getting the connection.

The `displayWarning` method creates a loop to display all warnings returned by calling the `getWarnings` method for the connection object. The `displayWarnings` method is defined in the script on page 52.

- [⑥](#) creates the `dma` object of the `DatabaseMetaData` class by calling the `java.sql.Connection.getMetaData` method for the connection object created in [⑤](#). The `getMetaData` method returns information such as the connected database's tables, supported SQL grammar, and stored procedures.

These lines also print url, driver name, and driver version information for the `dma` object, as returned by the `getURL`, `getDriverName`, and `getDriverVersion` methods in the `java.sql.DatabaseMetaData` class.

- [⑦](#) creates the statement object by calling the `java.sql.Connection.createStatement` method for the connection object created in [⑤](#). The statement will hold the SQL query statement to be executed for the connection.

- [⑧](#) creates a `resultSet` object of the `java.sql.ResultSet` class by calling the `java.sql.Statement.executeQuery` method for the `statement` object created in [⑦](#).

The `executeQuery` method executes the SQL statement using the query text as a parameter. The `resultSet` object provides access to the table of data generated by executing the query.

- In [⑨](#), if the `resultSet` object is not null, the program will display the returned data by calling the `displayResultSet` method (which is defined in the script on page 54.) If the `resultSet` object is null, the program prints *the result is empty*.
- [⑩](#) closes the `resultSet` object by calling the `java.sql.ResultSet.close` method, thereby releasing the database and JDBC resources. It also ends the `try` block.

```

public void run() {
①   String url = urlField.getText();
      String query = queryField.getText();
      String user = userField.getText();
      String pwd = pwdField.getText();

②   Connection connection = null;
      Statement statement = null;

      // to log all sql function calls into a file, uncomment this
      // block.
      // try {
③   //     DriverManager.setLogStream(new PrintStream(new
      //         FileOutputStream("jdbcdemo.log"));
      // }
      // catch(java.io.IOException IOEx)
      // {System.out.println("Cannot create log file."); }

④   try {

⑤       connection = DriverManager.getConnection (url, user,
          pwd);

          displayWarning(connection.getWarnings());
          connection.clearWarnings();

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

⑥       DatabaseMetaData dma = connection.getMetaData();

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

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

⑦       statement = connection.createStatement();

          // Submit a query, creating a ResultSet object

⑧       ResultSet resultSet = statement.executeQuery(query);

```

Back

```
        // Display all columns and rows from the result set
    ⑨ — [ if (resultSet != null)
          displayResultSet (resultSet);
          else
          println("The result is empty");

          // Close the result set
    ⑩ — [ resultSet.close()
          }
    ]
```

Back

Catching Exceptions Resulting from Query Execution

The next section of code catches and handles exceptions that might be generated while executing the SQL query.

- Because the `Connection`, `DatabaseMetaData`, `Statement`, and `ResultSet` class methods used in the `try` block can throw SQL exceptions, ① catches and prints `SQLException` objects (which are held in the `ex` variable created by the `java.sql.SQLException` constructor).

The `while` loop prints exception information returned by the `getSQLState`, `getMessage`, `getErrorCode`, and `getNextException` methods for each exception caught.

- ② catches and prints other types of exceptions thrown in the Java environment (held in the `ex` variable in the `java.lang.Exception` class) during execution of the `try` block.

```
catch (SQLException ex) {  
    // A SQLException was generated. Catch it and  
    // display the error information.  
    println ("\n*** SQLException caught ***\n");  
    while (ex != null) {  
        ① println("SQLState: " +  
            ex.getSQLState());  
        println("Message: " +  
            ex.getMessage());  
        println("Vendor: " +  
            ex.getErrorCode());  
        ex = ex.getNextException ();  
        println("");  
    }  
}  
catch (java.lang.Exception ex) {  
    ② // Got some other type of exception.  
    ex.printStackTrace();  
}
```

[Back](#)

Closing the Query and Ending the Run Method

The next section of code closes the SQL query, the database connection, and the run method definition.

- ① closes the statement object and the connection object by calling the `java.sql.Statement.close` and `java.sql.Connection.close` methods.
- ② catches and prints SQL exceptions and Java language exceptions generated while closing the statement and connection objects.
- ③ ends the run method definition for the `jdbcdemo` class.

```
try {  
    // Close the statement  
    statement.close();  
    // Close the connection  
    connection.close();  
}  
  
catch (SQLException ex {  
    // A SQLException was generated. Catch it and  
    // display the error information.  
  
    println ("\n*** SQLException caught ***\n");  
  
    while (ex != null) {  
        println("SQLState: " +  
            ex.getSQLState());  
        println("Message: " +  
            ex.getMessage());  
        println("Vendor: " +  
            ex.getErrorCode());  
        ex = ex.getNextException ();  
        println("");  
    }  
}  
catch (java.lang.Exception ex) {  
    // Got some other type of exception.  
    ex.printStackTrace();  
}  
}
```

[Back](#)

To summarize the `run` definition method, this method:

- Connects to the database specified in the application's URL field
- Creates a SQL statement to send to the connected database
- Executes the SQL statement and gets the results, using the text specified in the application's QUERY field as the statement text
- Displays the results in the application window's output area
- Processes warnings and exceptions that occur during database connection and query execution
- Closes the query and the connection

displayWarning Method Definition

The next section of code defines the `displayWarning` method of the `jdbcdemo` class. This method displays warnings that are generated when connecting to the SQL database. (This method is used in block ⑤ on page 47.)

- ① declares the `displayWarning` method, which uses the `sqlWarning` variable of the `java.sql.SQLException` class as a parameter.

These lines also create the boolean `warning` variable and assign it an initial value of `false`.

- In ②, if the `sqlWarning` variable is not null, but holds an object, the `warning` variable value becomes `true`. The program prints lines of `sqlWarning` values returned by calling the `sqlWarning.getSQLState`, `getMessage`, and `sqlWarning.getErrorCode` methods for the `sqlWarning` object. (The `getMessage` method is inherited from the `java.sql.SQLException` class.)

Then the `sqlWarning.getNextWarning` method moves the `sqlWarning` value to the next warning. The while loop is repeated for that object. When there are no more warnings, the while loop terminates.

- [③](#) returns the warning variable's value and ends the `displayWarning` method definition.

```
//-----  
// displayWarning  
// Displays warnings. Returns true if a warning existed  
//-----  
  
① {  
    public boolean displayWarning(SQLWarning sqlWarning)  
        throws SQLException  
    {  
        boolean warning = false;  
  
        // If a SQLWarning object was given, display the  
        // warning messages. Note that there could be  
        // multiple warnings chained together.  
  
        if (sqlWarning != null) {  
            println ("\n *** Warning ***\n");  
            warning = true;  
            while (sqlWarning != null) {  
                println ("SQLState: " +  
                    sqlWarning.getSQLState());  
                println ("Message: " +  
                    sqlWarning.getMessage());  
                println ("Vendor: " +  
                    sqlWarning.getErrorCode());  
                println ("");  
                sqlWarning = sqlWarning.getNextWarning  
            }  
        }  
        return warning;  
    }  
} ③
```

[Back](#)

displayResultSet Method Definition

The next section of code defines the `displayResultSet` method of the `jdbcdemo` class. This method displays the result set returned from the connected SQL database in response to executing the application's query. (This method is used in block [9](#) on page 48.)

- [1](#) declares the `displayResultSet` method of the `jdbcdemo` class, using the `resultSet` object of the `java.sql.ResultSet` class as a parameter. (Block [8](#) on page 47 creates the `resultSet` object.)
- [2](#) creates the integer variable `i`, which is used in `for` blocks of the method definition to calculate column data to print.
- [3](#) creates a `resultSetMetaData` object by calling the `ResultSet.getMetaData` method for the `resultSet` object. The `getMetaData` method returns the number, types, and properties of the `resultSet` object's columns.
- [4](#) creates the `numCols` variable by calling the `ResultSetMetaData.getColumnCount` method for the `resultSetMetaData` object. The `getColumnCount` method returns the number of columns in the result set. The `numCols` value is used with the `i` value to calculate column data to print.
- In [5](#), the `for` block prints the result set's column labels by calling the `ResultSetMetaData.getColumnLabel` method for the `resultSetMetaData` object. This method uses the column index number as a parameter to get column labels.

This block uses the `i` and `numCols` values and the `print` method created in block [1](#) on page 55 to print the column labels in the text area defined by the `outArea` object (which is created in [7](#) on page 43).

- ⑥ creates a loop to print rows in the result set by calling the `ResultSet.next` method for the `resultSet` object. (The next method moves the result set's cursor to the next row of data, starting at row 0.)

Using the same expression and `print` method as ⑤, this block prints the value of each column in each row of the result set as a Java string. The `ResultSet.getString` method gets the values for the `resultSet` object as Java strings.

```
//-----  
// displayResultSet  
// Displays all columns and rows in the given result set.  
//-----  
① {  
    public void displayResultSet(ResultSet resultSet)  
        throws SQLException  
    {  
    ②     int i;  
  
        // Get the ResultSetMetaData.  
  
    ③     ResultSetMetaData resultSetMetaData =  
        resultSet.getMetaData ();  
  
        // Get the number of columns in the result set  
  
    ④     int numCols = resultSetMetaData.getColumnCount ();  
  
        // Display column headings  
  
    ⑤     for (i=1; i<=numCols; i++) {  
            if (i > 1) print(", ");  
            print(resultSetMetaData.getColumnLabel(i));  
        }  
        println("");  
    }  
}
```

Back

```

// Display data, fetching until end of the result set
while (resultSet.next()) {
    // Loop through each column, getting the
    // column data and displaying
    for (i=1; i<=numCols; i++) {
        if (i > 1) print(", ");
        print(resultSet.getString(i));
    }
    println("");
    // Fetch the next result set row
}

```

⑥

[Back](#)

Print and Println Method Definitions

The next section of the script defines the `print` and `println` methods of the `jdbcdemo` class, which are used in other sections used to print values into the `outArea` `TextArea` object.

- The `print` method defined in ① prints string values into the `outArea` object by calling the `appendText` method, using the `str` variable in the `String` class as a parameter.
- The `println` method in ② prints string values and a new line into the `outArea` object by calling the `appendText` method.

```

① {
public void print(String str) {
    if (str != null)
        outArea.appendText(str);
}

② {
public void println(String str) {
    if (str != null)
        outArea.appendText(str) ;
        outArea.appendText("\n");
}
}

```

main Method Definition

The final section of the `jdbcdemo` application script defines the `main` method of the `jdbcdemo` class and ends the class definition.

As with other Java applications, this method is used to start up the program. It creates an instance of `jdbcdemo`, for which it creates, sizes, and shows an application window.

```
public static void main(String[] args) {
    Frame f = new jdbcdemo("jdbcdemo", 750, 600);
    f.resize(750, 600);
    f.show();
}

//class jdbcdemo
}
```

Compiling `jdbcdemo`

To compile `jdbcdemo`, type the appropriate compile syntax shown in the following table (where `[jdk1.1]` is the directory where JDK1.1.

For:	Type:
Windows	<code>javac -classpath C:[jdk1.1]\lib\classes.zip;..\classes;.\jdbc\classes;. jdbcdemo.java</code>
UNIX	<code>javac -classpath [jdk1.1]/lib/classes.zip:../classes:./jdbc/classes;. jdbcdemo.java</code>

Creating a JDBC NetServer Applet

The process for creating an applet that uses JDBC NetServer is similar to the process for creating an application that uses JDBC NetServer. JDBC NetServer applets are built with Java packages, constructors, classes, and methods, as are JDBC NetServer applications. And for database connectivity, applets use the constructors, classes, and methods from the `agave.sql` and `java.sql` packages. Like applications, applet source code includes instructions to load and register the `AgaveJdbcDriver` for NetServer so that `agave.sql` classes are used for SQL calls.

However, when developing an applet you must derive the applet's class from the `Applet` class (versus from the `Frame` class for an application), and make additional changes as well.

In summary, to build a JDBC NetServer applet, use the same process you would use to create an application. However, change the process by taking the following steps to ensure that the program runs as an applet:

- Make sure that the NetServer (or the NetBridge) is running on the same machine as your Web server.
- When building the applet, derive the applet class, constructors, and methods from the `Applet` class; use the `init` function in place of the class constructor; and take other key steps to establish the program as an applet rather than an application.
- Determine whether to archive the applet classes and files.
- Bundle the applet and all supporting files and move them to the Web server for distribution.

Running NetServer or NetBridge on Web Server Machine

Remember that most browser applet restrictions prevent the applet from establishing a network connection with any machine other than the Web browser from which the applet was downloaded. This means that applet implementations must have the NetServer (or JDBC NetBridge) running on the same machine as the Web server.

This does not necessarily mean that the database must also be running on the same machine as the Web server. The NetServer can access many types of databases over a network using the database-proprietary network protocol (for databases that have a network protocol). In cases where this is not possible, applets can access the the database with NetBridge on the Web server. (See [JDBC NetBridge](#) in the [JDBC NetServer Overview](#) chapter for details about NetBridge).

Building an Applet from an Application

When building an applet, to distinguish it as an applet rather than an application, you must derive the applet's main class from the `Applet` class rather than the `Frame` class. You must also take additional steps, too, such as removing the `import java.sql` directive and using the `init` function rather than a class constructor to build the user interface.

After building the applet source code, you must compile it using the appropriate syntax. Also, for end-users to run the applet from a Web browser, an HTML file must reference the applet file using the `<APPLET>` tag.

Building the Applet Script

Following are the steps to build an applet that uses JDBC NetServer (named `myApplet`), based on an existing application.

1. Add the following directive before the class declaration:

```
import java.applet.*;
```

(The `import agave.sql.*` directive should already be included in JDBC NetServer application source. Add this directives if it is not.)
2. Remove the `import java.sql.*` directive before the class declaration. For Agave JDBC NetServer applets, all `java.sql` interfaces and classes are provided in the `agave.sql` package of classes. Therefore, including this directive in your applet source code results in a naming conflict.

Including `java.sql` interfaces and classes in the `agave.sql` package for applets enables applets to run on Web servers that have JDK 1.02 installed (which does not include `java.sql` as a standard JDK class package in the way that JDK 1.1 does).

3. In the class declaration, derive the class from the `Applet` class rather than the `Frame` class. For example:

```
public class myApplet extends Applet
```

4. Use the `init()` function rather than a constructor method after the class variable declarations to build the user interface. The browser calls the `init ()` function to create an object of the applet class.

For example, where the `myApplet` application might have the constructor: `myApplet(string param, int count);`

the `myApplet` applet should have the function: `init()`.

Note that, unlike a constructor, `init()` can take no parameters.

5. Set the layout manager in the `init` function to `BorderLayout` if you want to create a window like the application window. For example:

```
public void init() {  
    setLayout(new BorderLayout() );  
}
```

This is necessary because, although the application's frame defaults a window created by `BorderLayout`, the applet defaults to panel created with `FlowLayout`. (So although the `setLayout` method is not needed for the application, it is needed for the applet to override the implicit `FlowLayout`.)

6. Omit or comment out the `main` method, which usually contains code to make and size a new `frame` object. This method is not needed, because the browser makes an object of the class specified in the `<APPLET>` tag and also sets the applet panel size as specified in the tag.
7. Omit or comment out the `setTitle` method, which sets the application title bar, if it is used in the application. Applets do not have title bars. (You can set the Web page title bar with the `<TITLE>` tag in the corresponding HTML file.)

Compiling the JDBC NetServer Applet

To compile your JDBC NetServer applet, type the appropriate compile syntax shown in the following table (where `[jdk1.1]` is the directory where JDK1.1 is installed and `[myApplet]` is your applet name).

For:	Type:
Windows	<pre>javac -classpath C:\[jdk1.1]\lib\classes.zip;.\classes;. [myApplet].java</pre>
UNIX	<pre>javac -classpath /[jdk1.1]/lib/classes.zip:./classes:. [myApplet].java</pre>

Linking to the Applet with the HTML <APPLET> Tag

The HTML file that runs your JDBC NetServer applet must be located in the same directory as your applet. Also, to be available on the World Wide Web, these files, along with the classes used to build the applet, must on your Web server in the same directory.

As mentioned previously, the HTML file that references your applet must use the <APPLET> tag to do so. This tag is an extension to HTML for including applets in Web pages. It requires an end tag.

Following are examples of APPLET element blocks for a non-archived and an archived applet file:

```
<APPLET CODE="NonArchivedApplet.class" WIDTH=750  
HEIGHT=600>  
</APPLET>
```

```
<APPLET ARCHIVE="appletdemo.zip"  
CODE="appletdemo.class" WIDTH=750 HEIGHT=600>  
</APPLET>
```

The basic <APPLET> attributes are CODE, WIDTH, and HEIGHT, which are required to identify the applet and set its boundaries on the screen. The ARCHIVE attribute is required for archived applets.

- CODE identifies the applet file name. The file name must include the .class extension and be in the same directory as the referencing HTML file.
- WIDTH and HEIGHT indicate, in pixels, the size of the window that will hold the applet.
- ARCHIVE identifies the name of the archive file that includes the applet file. This file must be in the same directory as the referencing HTML file.

Following is the HTML source for the JDBC NetServer appletdemo:

```
<HTML>
<TITLE> applet demo</TITLE>
<BODY>
Here is the applet.
<APPLET ARCHIVE="appletdemo.zip"
CODE="appletdemo.class" WIDTH=750 HEIGHT=600>
</APPLET>
</BODY>
</HTML>
```

Determining Whether to Archive Applet Classes

Before you distribute your applet, you must decide whether to archive your applet or distribute it as separate classes.

If you distribute the applet as a *.zip archive, you must make sure your client browser understands the `<APPLET ARCHIVE="myApplet.zip" . . . >` attribute. Currently Netscape and Internet Explorer support this attribute.

Unfortunately, neither browser supports caching of large archives. Therefore, your archive must be downloaded each time the browser is restarted. This download time is relatively insignificant on a large network. However, the download can require several minutes over a dial-up connection.

With classes distributed separately, the browser will archive each of the classes. This decreases the start-up time over a slower network (although the same several minutes are required as for an archive the first time the class is downloaded). Unfortunately, over a faster network, distributing your application as separate classes might slow the applet startup time due to time required by the browser to verify each of the classes.

Generally, it is a good idea to archive your applet if you are on a fast network and your browser supports archiving. If you are on a slower network or your browser does not support archiving, distribute your applet as separate classes.

Distributing Applet as a *.zip Archive

If you choose to archive your applet, first obtain a copy of the zip/unzip application. It is available for Windows 95/NT from *ftp://ftp.simtel.net/pub/simtelnet/win3/compress/pk250w16.exe*. This application is distributed shareware, and the authors expect you to purchase it if you use it. (Versions of zip/unzip are also available for many UNIX platforms.)

Use the zip command to archive all of the classes in a directory into a *.zip file. The following table shows an example zip command for UNIX and Windows:

For	Example Zip Command
UNIX	zip -0 applet.zip *.class
Windows	zip -0 applet.zip *.cla

Remember that **applet archives (*.zip files) must not be compressed!** Use no path information and store only options.

The example applet archive can be referenced from an HTML file with the following syntax (with the XX Xvalue specified in pixels):

```
<APPLET ARCHIVE=applet.zip CLASS="myappl.class"  
WIDTH=XXX HEIGHT=XXX>
```

Distributing Separate Applet Class Files

If you choose to distribute your applet as separate classes, simply copy your applet class file, along with the files in the NetServer/client/applet/classes directory, into the same directory on your Web server.

The applet can then be referenced from an HTML file using the following syntax (with the XXX value specified in pixels):

```
<APPLET CLASS="myappl.class" WIDTH=XXX  
HEIGHT=XXX>
```

Distributing the Applet and Supporting Files on the Web server

To make your applet available on the Web, you must bundle it in the same directory with the referencing HTML file and all supporting classes files, and copy that directory on your Web server. To distribute your applet:

1. Change to the NetServer\appletdemo (or NetServer/appletdemo) directory, and create a new directory (for example, *myAppletStuff*) for your applet files.
2. Copy all files from the NetServer\appletdemo\classes (or NetServer/appletdemo/classes) directory into the myAppletStuff directory.
3. If you are zipping applet files, zip the whole myAppletStuff directory into the myApplet.zip file. Use no path information and the zip option of store. (Do **not** use not archive!)
4. Place the complete, unzipped myAppletStuff directory (with all classes separate) or the myApplet.zip file on the Web server.
5. Place the myApplet.html file that references the applet in the same directory with the myApplet file(s) on the Web server.

JDBC NetServer Applet Demo Script

Similar in many ways to the `jdbcdemo.java` application script, the sections of the `appletdemo.class` script perform the following tasks:

- [Import Java language packages to be used by the program](#)
- [Create the `appletdemo` class, class instance variables, and user interface \(but using the `init` function rather than a constructor method\)](#)
- [Process the selected user action](#)
- [Run methods that execute the user's query, catch resulting exceptions, and close the query connection](#)
- [Define class methods used to perform requested functions throughout the applet script](#)

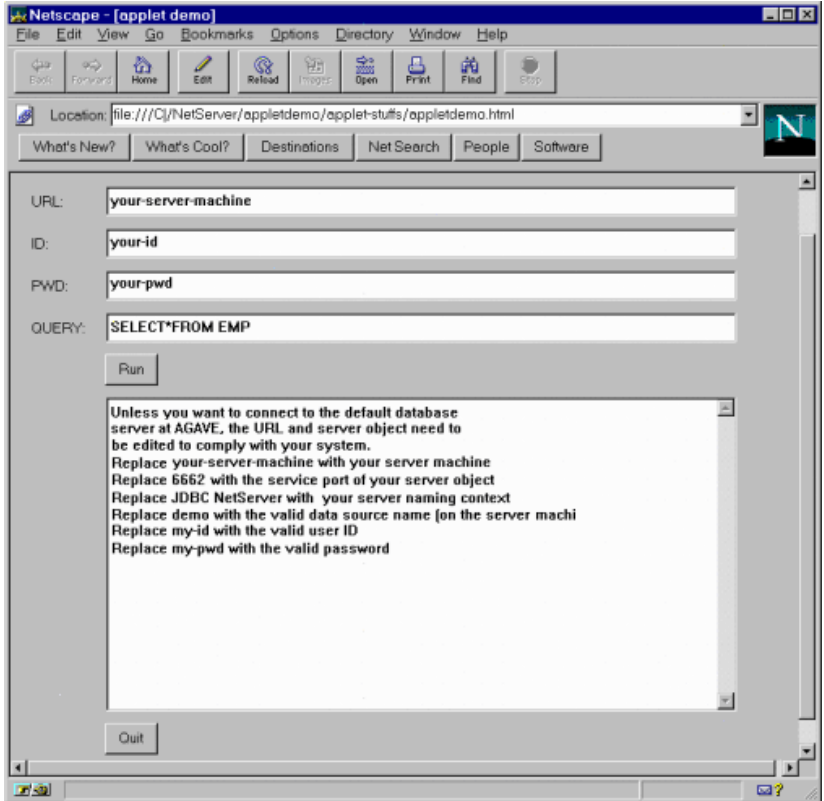
You can view the `appletdemo` source code online by opening the `appletdemo.class` file in the appropriate location on your machine. And you can view the applet [user interface](#) by opening the `appletdemo.html` file in the appropriate system directory from your Web browser.

The following table identifies the paths of these files:

For	The <code>appletdemo.class</code> and <code>appletdemo.html</code> path are:
Windows 95/NT	<ul style="list-style-type: none"> • <code>[InstallDir]\NetServer\appletdemo\applet-stuffs\appletdemo.class</code> • <code>[InstallDir]\NetServer\appletdemo\applet-stuffs\appletdemo.html</code>
UNIX	<ul style="list-style-type: none"> • <code>[InstallDir]/NetServer/appletdemo/applet-stuffs/appletdemo.class</code> • <code>[InstallDir]/NetServer/appletdemo/applet-stuffs/appletdemo.html</code>

jdbcdemo Application User Interface

Following is an illustration of the jdbcdemo application user interface, which enables end-users to query a database from their web browsers:



Importing Java Packages

The following statements import the packages of java code used by the appletdemo applet. Notice that this section includes the `import java.applet.Applet;` directive to import the `Applet` class from the `java.applet` package. It also omits the `import java.sql.*` directive.

```
import agave.sql.*;
import java.io.*;
import java.net.URL;
import java.awt.*;
import java.applet.Applet;
```

Creating appletdemo Class, Class Variables, and User Interface

The next section of the appletdemo script creates the appletdemo class and class variables. Notice the following:

- In [①](#), a commented `java.lang.ClassLoader` constructor can be uncommented to request the class loader to load a class with the specified name.
- In [②](#), the class constructor derives the class from the `java.applet.Applet` class.

Notice that the driver variable is set to null in the variable declarations.

- In [③](#), the `init ()` method is used to initialize the class, enabling the Web browser to build the applet user interface.
- In [④](#), the `setLayoutManager` method sets the layout manager to `BorderLayout`. Without this setting, the applet would not be displayed in a window like the application.

- In ⑤, notice the instructions to load and initialize the new AgaveJdbcDriver.

```
//-----  
//  
//File:      appletdemo.java  
//  
//Description: Program to test and demonstrate performance of  
//JDBC NetServer.  
//  
//Date:      January 25, 1997  
//  
//Programmer: Glenn Deitiker  
//  
//-----  
① //class IncludedClassLoader extends ClassLoader {  
//    public IncludedClassLoader() {}  
//    public synchronized Class loadClass(String name,  
//        boolean tf ) {  
//        return null;  
//    }  
// }  
② public class appletdemo extends Applet {  
  
    String url;  
    String user;  
    String pwd;  
    String sql;  
    Driver driver;  
    Button runButton;  
    Button quitButton;  
    TextField urlField;  
    TextField userField;  
    TextField pwdField;  
    TextField queryField;  
    TextArea outArea;  
    Label label;
```

[Back](#)

```
③ public void init() {  
④     setLayout(new BorderLayout() );  
  
    label = new Label("URL:");  
    add(label);  
    label.reshape(10, 30, 60, 30);  
  
    urlField = new TextField  
    ("your-server-machine:6662/JDBCNetServer/jdbc:demo");  
    add(urlField);  
    urlField.reshape(80, 30, 600, 30);  
  
    label = new Label("ID:");  
    add(label);  
    label.reshape(10, 70, 60, 30);  
  
    userField = new TextField("your-id");  
    add(userField);  
    userField.reshape(80, 70, 60, 30);  
  
    label = new Label("PWD:");  
    add(label);  
    label.reshape(10, 110, 60, 30);  
  
    pwdField = new TextField("your-pwd");  
    add(pwdField);  
    pwdField.reshape(80, 110, 600, 30);  
  
    label = new Label("QUERY:");  
    add(label);  
    label.reshape(10, 150, 60, 30);  
  
    queryField = new TextField("SELECT * FROM EMP");  
    add(queryField);  
    queryField.reshape(80, 150, 600, 30);  
  
    runButton = new Button("Run");  
    add(runButton);  
    runButton.reshape(80, 190, 50, 30);
```

[Back](#)

```
outArea = new TextArea("", 5 60);
add(outArea);
outArea.reshape(80, 230, 600, 300);
println("Unless you want to connect to the default
        database");
println("server at AGAVE, the url and server object need");
println("to be edited to comply with your system.");
println("Replace your-server-machine with your server
        machine.");
println("Replace 6662 with the service port of your server
        object.");
println("Replace JDBCNetServer with your server naming
        context.");
println("Replace demo with the valid data source name(on
        the server machine).");
println("Replace your-id with the valid user ID.");
println("Replace your-pwd with the valid password.");

quitButton = new Button("Quit");
add(quitButton);
quitButton.reshape(80, 540, 50, 30);
```

[Back](#)

```
⑤- driver = new AgaveJdbcDriver();

    try {

        driver = new AgaveJdbcDriver();
        DriverManager.registerDriver(driver);
    }
```

```
catch (SQLException ex) {

    //A SQLException was generated. Catch it and
    // display the error information.

    println ("\n*** SQLException caught ***\n");

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

    // Got some other type of exception.

    ex.printStackTrace();

}

}
```

[Back](#)

Processing the User Action

The following source code used to process the appletdemo **Quit** or **Run** action is the same as for the jdbcdemo application.

```
public boolean action(Event e, Object arg)
{
    if (e.target instanceof Button) {
        if ("Quit".equals(arg))
        {
//            this.dispose();
            System.exit(0);
        }

        if ("Run".equals(arg))
        {
            run();
        }
    }
    return false;
}
```

Running the Program and Ending the Run Method

The block of code used to run the appletdemo applet and execute SQL queries is the same as for the jdbcdemo application. However, the appletdemo applet provides methods not in the jdbcdemo source to do the following:

- Calculate and the print before and after connection times, in milliseconds, as shown in [①](#).
- Calculate and the print before and after query times, in milliseconds, as shown in [②](#).
- Print the size of the result set, as shown in [③](#), which is calculated by the `sizeResultSet` method defined later in the script.

```
public void run() {

    String url = urlField.getText();
    String query = queryField.getText();
    String user = userField.getText();
    String pwd = pwdField.getText();

    Connection connection = null;
    Statement statement = null ;

    // to log all sql function calls into a file, uncomment this
    // block.
    // try {
    //     DriverManager.setLogStream(new PrintStream(new
    //         FileOutputStream("jdbcdemo.log")));
    // }
    // catch(java.io.IOException IOEx)
    // {System.out.println("Cannot create log file."); }

    try {
        ① long tBeforeConnect = System.currentTimeMillis();
        connection = DriverManager.getConnection (url, user,
            pwd);
        long tAfterConnect = System.currentTimeMillis();

        print("\nConnect time(ms): ");
        println( tAfterConnect-tBeforeConnect );

        displayWarning(connection.getWarnings());
        connection.clearWarnings();

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

        DatabaseMetaData dma = connection.getMetaData ();

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

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

        statement = connection.createStatement();
```

[Back](#)

```
        // Submit a query, creating a ResultSet object
    ②    long tBeforeQuery = System.currentTimeMillis();
        ResultSet resultSet = statement.executeQuery(query);
        long tAfterQuery = System.currentTimeMillis();

        print("\nQuery time & data xfer(ms): ");
        println( tAfterQuery-tBeforeQuery );
        println("");

    ③    //print("\nSizeof result set: ");
        //println( sizeResultSet( resultSet ) );

        // Display all columns and rows from the result set
        if (resultSet != null)
            displayResultSet (resultSet);
        else
            println("The result is empty");

        // Close the result set

        resultSet.close()

    }

    catch (SQLException ex) {

        // A SQLException was generated. Catch it and
        // display the error information.

        println ("\n*** SQLException caught ***\n");

        while (ex != null) {
            println("SQLState: " +
                ex.getSQLState());
            println("Message: " +
                ex.getMessage());
            println("Vendor: " +
                ex.getErrorCode());
            ex = ex.getNextException ();
            println("");
        }
    }
}
```

[Back](#)

```
catch (java.lang.Exception ex) {
    // Got some other type of exception.
    ex.printStackTrace();
}
try {
    // Close the statement
    statement.close();
    // Close the connection
    connection.close();
}
catch (SQLException ex {
    // A SQLException was generated. Catch it and
    // display the error information.
    println ("\n*** SQLException caught ***\n");
    while (ex != null) {
        println("SQLState: " +
            ex.getSQLState());
        println("Message: " +
            ex.getMessage());
        println("Vendor: " +
            ex.getErrorCode());
        ex = ex.getNextException ();
        println("");
    }
}
catch (java.lang.Exception ex) {
    // Got some other type of exception.
    ex.printStackTrace();
}
}
```

[Back](#)

Method Definitions

The final section of the `appletdemo` applet script provides method definitions for new methods of the `appletdemo` class, which are used in the previous sections. The `appletdemo` uses these methods to display result sets returned from a SQL database and to print information, similar to the `jdbcdemo` applet. The `appletdemo` method declarations also define methods used to:

- Calculate the size of a result set, as shown in [①](#).
- Print a line with a long value in the output area, as shown in [②](#).

Notice that the `main` declaration at the end of the `appletdemo` script is commented out, as shown in [③](#).

```
//-----  
// displayWarning  
// Displays warnings. Returns true if a warning existed  
//-----  
  
public boolean displayWarning(SQLWarning sqlWarning)  
    throws SQLException  
{  
    boolean warning = false;  
  
    // If a SQLWarning object was given, display the  
    // warning messages. Note that there could be  
    // multiple warnings chained together.  
  
    if (sqlWarning != null) {  
        println ("\n *** Warning ***\n");  
        warning = true;  
        while (sqlWarning != null) {  
            println ("SQLState: " +  
                sqlWarning.getSQLState());  
            println ("Message: " +  
                sqlWarning.getMessage());  
            println ("Vendor: " +  
                sqlWarning.getErrorCode());  
            println ("");  
            sqlWarning = sqlWarning.getNextWarning  
        }  
    }  
    return warning;  
}  
  
//-----  
// displayResultSet  
// Displays all columns and rows in the given result set.  
//-----  
  
public void displayResultSet(ResultSet resultSet)  
    throws SQLException  
{  
    int i;  
  
    // Get the ResultSetMetaData.  
  
    ResultSetMetaData resultSetMetaData =  
        resultSet.getMetaData ();
```

[Back](#)

```
        // Get the number of columns in the result set
        int numCols = resultSetMetaData.getColumnCount ();

        // Display column headings
        for (i=1; i<=numCols; i++) {
            if (i > 1) print(", ");
            print(resultSetMetaData.getColumnLabel(i));
        }
        println("");

        // Display data, fetching until end of the result set
        while (resultSet.next()) {

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

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

            // Fetch the next result set row

        }
    }
```

[Back](#)

①

```
//-----
// sizeResultSet
// Calculate size of the result set.
//-----

public int sizeResultSet(ResultSet resultSet)
    throws SQLException
{
    int i;
    int size=0;

    // Get the ResultSetMetaData.

    ResultSetMetaData resultSetMetaData =
        resultSet.getMetaData ();
```

```
// Get the number of columns in the result set
int numCols = resultSetMetaData.getColumnCount ();

// Display column headings
for (i=1; i<=numCols; i++) {
    if ( resultSetMetaData.getColumnLabel(i) != null )
    {
        size+=resultSetMetaData.getColumnLabel(i).length();
    }
}

// Display data, fetching until end of the result set
while (resultSet.next())
{
    // Loop through each column, getting the
    // column data and displaying

    for (i=1; i<=numCols; i++) {
        if ( resultSet.getString(i) != null )
        {
            size+=resultSet.getString(i).length();
        }
    }
    // Fetch the next result set row
}
return( size );
}

public void print(String str) {
    if (str != null)
        outArea.appendText(str);
}

public void println(String str) {
    if (str != null)
        outArea.appendText(str);
        outArea.appendText("\n");
}
}
```

[Back](#)

```
② public void println(long value) {  
    outArea.appendText( java.lang.Long.toString(value) );  
    outArea.appendText( "\n" );  
}
```

[Back](#)

```
③ // public static void main (String[] args) {  
//  
//     IncludedClassLoader loader = new IncludedClassLoader();  
//     Object main = loader.loadClass(  
//         "java/sql/SQLException",true );  
//  
//     Frame f = new appletdemo("appletdemo", 750, 600);  
//     this.resize(750, 600);  
//     this.show();  
// }  
}  
  
//class appletdemo
```