ACTIAN™

JDBC Driver Guide

Developing Applications Using the Zen JDBC Driver

Zen v16

Activate Your Data™

Copyright © 2024 Actian Corporation. All Rights Reserved.

This Documentation is for the end user's informational purposes only and may be subject to change or withdrawal by Actian Corporation ("Actian") at any time. This Documentation is the proprietary information of Actian and is protected by the copyright laws of the United States and international treaties. The software is furnished under a license agreement and may be used or copied only in accordance with the terms of that agreement. No part of this Documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or for any purpose without the express written permission of Actian. To the extent permitted by applicable law, ACTIAN PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, AND ACTIAN DISCLAIMS ALL WARRANTIES AND CONDITIONS, WHETHER EXPRESS OR IMPLIED OR STATUTORY, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OR OF NON-INFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT WILL ACTIAN BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF ACTIAN IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The manufacturer of this Documentation is Actian Corporation.

For government users, the Documentation is delivered with "Restricted Rights" as set forth in 48 C.F.R. Section 12.212, 48 C.F.R. Sections 52.227-19(c)(1) and (2) or DFARS Section 252.227-7013 or applicable successor provisions.

Actian, Actian DataCloud, Actian DataConnect, Actian X, Avalanche, Versant, PSQL, Actian Zen, Actian Director, Actian Vector, DataFlow, Ingres, OpenROAD, and Vectorwise are trademarks or registered trademarks of Actian Corporation and its subsidiaries. All other trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Contents

About This Document

This documentation covers development of Zen applications that use the Java API for executing SQL statements (JDBC).

Who Should Read This Manual

This documentation is designed for the user who is developing Zen applications using the Java API for executing SQL statements (JDBC).

We would appreciate your comments and suggestions about this document. Your feedback can determine what we write about the use of our products and how we deliver information to you. Please post your feedback in the community forum on the Actian Zen website.

Introducing the Zen JDBC Driver

The following topics introduce you to the Zen Java Database Connectivity (JDBC) interface:

- Zen JDBC Support
  - JDBC Requirements
  - JDBC Features
  - Zen JDBC Data Types
- Zen JDBC Driver Limitations
  - Unsupported APIs
  - Driver Limitations

For instructions and details on this Zen feature, see Programming with the Zen JDBC 2 Driver and JDBC API Reference.

Zen JDBC Support

JDBC is a standard API that Java programmers can use to develop database and Internet applications using Java. It consists of interfaces to develop SQL based database applications in the Java programming language. The JDBC interfaces are included as part of the Java Developer Kit.

JDBC is the counterpart of ODBC in Java and is heavily influenced by ODBC and relational databases.

Detailed information on the JDBC API is available at the Oracle website.

The rest of this section covers the following topics:

JDBC Requirements

The Zen JDBC driver works in conjunction with Zen. You can use the Enterprise Server, Cloud Server, or a Workgroup engine.

JDBC Features

The following list summarizes features of the Zen JDBC driver:

- 100% Java certified
- JDBC 4 compliant, type 4 driver that is also compatible with applications that use JDBC 2 drivers
- Supports thread safe operation
- Supports transactions isolation levels supported by the Zen engine, for example READ_COMMITTED, serializable
- Performs result set caching to reduce network access
- Supports binary data through the longvarbinary data type (2 GB limit)
- Supports long char data through the longvarchar and nlongvarchar data types (2 GB limit)
- Supports stored procedures with parameters
- Encrypts connection strings to provide security

- Support for code page filtering when reading from the database by specified the code page using a connection string parameter
- Support for result set cursors CONCUR_UPDATABLE, TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE
- Supports the DataSource interface to register Zen databases in JNDI, shielding your applications from specific driver features for Zen
- Supports the ParameterMetaData interface

Zen JDBC Data Types

The numerical identifiers used to specify Zen JDBC data types in some cases differ from the JDBC standard identifiers. The following table gives the full list.

Data Type	Identifier
AUTOTIMESTAMP	93
BFLOAT4	7
BFLOAT8	8
BIGIDENTITY	-5
BIGINT	-5
BINARY	-2
BIT	-7
CHAR	1
CURRENCY	3
DATE	91
DATETIME	93
DECIMAL	3
DOUBLE	8
IDENTITY	4
INTEGER	4
LONGVARBINARY	-4
LONGVARCHAR	-1

Data Type	Identifier
MONEY	3
NCHAR	-15
NLONGVARCHAR	-10
NUMERIC	2
NVARCHAR	-9
REAL	7
SMALLIDENTITY	5
SMALLINT	5
TIME	92
TIMESTAMP	93
TIMESTAMP2	93
TINYINT	-6
UBIGINT	-5
UINTEGER	4
USMALLINT	5
UTINYINT	-6
VARCHAR	12

Data Type	Identifier
MONEY	3

Data Type	Identifier

Zen JDBC Driver Limitations

Unsupported APIs

The Zen JDBC driver does not support the following JDBC interfaces:

- Array
- Blob
- Clob
- NClob
- Ref
- RowId
- SQLXML
- Struct
- SQLData
- SQLInput
- SQLOutput
- URL

These are not supported due to the fact the Zen engine does not currently support the underlying SQL 3 data types.

Driver Limitations

- You cannot use long data in out parameters.
- The smallest actual fetch size is two rows.
- You cannot have an updatable result set with a join.
- You cannot have an updatable result set with a GROUP BY.
- The JDBC driver will not store data in UnicodeBig or UnicodeLittle formats.
- The only Holdability is HOLD_CURSORS_OVER_COMMIT.
- Pooled statements are not supported.
- Named parameters are not supported.

Programming with the Zen JDBC 2 Driver

The following topics give an overview of using Zen with JDBC 2.0:

- How to Set Up Your Environment
- JDBC Programming Tasks
- Developing Web-based Applications
- JDBC 2.0 Standard Extension API
- Connection and Concurrency
- Scrollable Result Sets
- JDBC Programming Sample

How to Set Up Your Environment

This topic contains information about proper configuration for use of the JDBC interface.

- Setting the CLASSPATH System Variable
- Setting the PATH System Variable
- Loading the JDBC Driver into the Java Environment
- Specifying a Data Source
- Developing JDBC Applets

An online tutorial called Accessing Actian Zen with Java and JDBC is also available.

Setting the CLASSPATH System Variable

So that Java applications and applets recognize the Zen JDBC Driver, set your CLASSPATH environment variable to include the pvjdbc2.jar, pvjdbc2x.jar, and jpscs.jar files. By default, these files are installed on Windows platforms in the *install_directory*\bin folder under Program Files. On Linux and Raspbian, the files are installed by default to /usr/local/actianzen/bin/lib.

From Windows

```
set CLASSPATH=%CLASSPATH%;<path to pvjdbc2.jar directory>/pvjdbc2.jar
set CLASSPATH=%CLASSPATH%;<path to pvjdbc2x.jar directory>/pvjdbc2x.jar
set CLASSPATH=%CLASSPATH%;<path to jpscs.jar directory>   /jpscs.jar
```

To make the change permanent, edit the environment variables in the Windows System settings.

From Linux

```
export CLASSPATH=$CLASSPATH:<path to pvjdbc2.jar directory>/pvjdbc2.jar
export CLASSPATH=$CLASSPATH:<path to pvjdbc2x.jar directory>/pvjdbc2x.jar
export CLASSPATH=$CLASSPATH:<path to jpscs.jar directory>/jpscs.jar
```

Setting the PATH System Variable

If you connect to the database engine using shared memory, the JDBC driver must find pvjdbc2.dll or w64pvjdbc2.dll. Make sure that your PATH variable on Windows contains the location of the DLL:

```
set PATH=%PATH%;<path to pvjdbc2.dll directory or path to w64pvjdbc2.dll directory>
```

If you connect to the database engine using sockets, no DLL is required. Make sure that the version of pvjdbc2.dll or w64pvjdbc2.dll in your path matches the version of the .jar files in your CLASSPATH.

Loading the JDBC Driver into the Java Environment

After setting the CLASSPATH, you can now reference the Zen JDBC Driver from your Java application. You do this by using the java.lang.Class class:

```
Class.forName("com.pervasive.jdbc.v2.Driver");
```

IPv6 Environments

If you want to use the Zen JDBC driver in an IPv6-only environment, we recommend that you also use Java JRE 1.7. You may encounter issues with license counts or client-tracking problems if your application uses Java JRE 1.6 or earlier in an IPv6-only environment.

You may also encounter issues with license counts for the following combination of conditions:

1. A machine runs multiple applications using the Zen JDBC driver and the applications connect to the database engine with a combination of IPv4 and IPv6 addresses.
2. The SYSTEM PATH on the machine does **not** include the location of pvjdbc2.dll or w64pvjdbc2.dll. See also Setting the PATH System Variable.

Specifying a Data Source

After loading the PervasiveDriver class into your Java environment, you need to pass a URL-style string to the java.sql.DriverManager class to connect to a Zen database. The syntax for URL for the JDBC driver is as follows:

```
jdbc:pervasive://<machinename>:<portnumber>/<datasource>
```

<machinename>	is the host name or IP address of the machine that runs the database engine.
<portnumber>	is the port on which the database engine is listening. By default it is 1583.
<datasource>	is the name of the ODBC DSN on the database server that the application intends to use.

For example, if your Zen engine is on a machine named DBSERV, and you wish to connect to the Demodata database, your URL would look like this (assuming the server is configured to use the default port):

```
jdbc:pervasive://dbserv/demodata
```

To connect to the database using the DriverManager class, use the syntax:

```
Connection conn = DriverManager.getConnection("jdbc:pervasive://dbserv:1583/demodata", loginString,
passwordString);
```

where *loginString* is the string for a user login and *passwordString* is the string for the user password.

Note: The Zen engine must run on the specified host for JDBC applets and applications to access data.

Developing JDBC Applets

To develop web-based applications using JDBC, you need to place the JDBC .jar file in the code base directory containing the applet classes.

For example, if you are developing an application called MyFirstJDBCapplet, you need to place the pvjdbc2.jar file in the directory containing the MyFirstJDBCapplet class. For example, it might be C:\inetpub\wwwroot\myjdbc\. This enables the client web browser to download the JDBC driver and connect to the database.

You also need to put the archive parameter within the <applet> tag. For example:

```
<applet CODE="MyFirstJDBCapplet.class"
    ARCHIVE="pvjdbc2.jar" WIDTH=641 HEIGHT=554>
```

Note that the Zen engine must be running on the web server that hosts the applet.

JDBC Programming Tasks

This section highlights important concepts for JDBC programming.

Connection String Overview

The JDBC driver requires a URL to connect to a database. The JDBC driver uses the following URL syntax:

jdbc:pervasive://*machinename*:*port number*/*datasource*[;encoding=;encrypt=;encryption=]

machinename is the host name or ip address of the machine that runs the Zen server.

port number is the port on which the Zen server is listening. By default it is 1583.

datasource is the name of the ODBC engine data source on the Zen server that the application intends to use.

encoding= is the character encoding, which allows you to filter data you read through a specified code page so that it is formatted and sorted correctly. The value "auto" will determine the database code page at connection time and then set the encoding to that character encoding. The value "auto" will also preserve NCHAR literals in SQL queries. If not "auto," then SQL queries are converted to the database code page.

encrypt= specifies whether the JDBC driver should use encrypted network communications, also known as wire encryption.

encryption= specifies the minimum level of encryption allowed by the JDBC driver.

Note: A Zen engine needs to be running on the specified host to run JDBC applications.

Connection String Elements

The following configuration information and table of connection string elements show how to connect to a Zen database using JDBC:

Driver Classpath

```
com.pervasive.jdbc.v2
```

Statement to Load Driver

```
Class.forName("com.pervasive.jdbc.v2.Driver");
```

URL

```
jdbc:pervasive://server:port/DSN[;encoding=;encrypt=;encryption=]
```

or

```
jdbc:pervasive://server:port/DSN[?pvtranslate=&encrypt=&encryption=]
```

Argument	Description
server	The server name using an ID or a URL.
port	The default port for the Relational Engine is 1583. If no port is specified, the default is used.
DSN	Name of the DSN to set up on the server using regular ODBC methods.
encoding	See Using Character Encoding
encrypt	Determines whether the JDBC driver should use encrypted network communications, also known as wire encryption. (See Wire Encryption in *Advanced Operations Guide*.) Values: **always**, **never** If this option is not specified, the driver reflects the server's setting, the equivalent of the value "if needed." If the value **always** is specified, the JDBC driver uses encryption or else return an error if wire encryption is not allowed by the server. If the value **never** is specified, the JDBC driver does not use encryption and returns an error if wire encryption is required by the server. To use wire encryption with the JDBC driver, another JAR file is required to be in your class path. This JAR, jpscs.jar, is installed by default and uses Java Cryptography Extensions (JCE).
encryption	Determines the minimum level of encryption allowed by the JDBC driver. Values: **low**, **medium**, **high** Default: **medium** These values correspond to 40-bit, 56-bit, and 128-bit encryption, respectively. The following example specifies that the JDBC driver uses UTF-8 encoding, always requires encryption and requires at least the low level of encryption or it returns an error code. `jdbc:pervasive://host/demodata?encoding=UTF-8&encrypt=always&encryption=low`

JDBC Connection String Example

The following code is an example of connecting to a Zen database using the JDBC driver:

```
// Load the JDBC driver
Class.forName("com.pervasive.jdbc.v2.Driver")

// JDBC URL Syntax:
// jdbc:pervasive://<hostname or ip address > :
//   <port num (1583 by default)>/<odbc engine DSN>

String myURL = "jdbc:pervasive://127.0.0.1:1583/demodata";
 try
    {

// m_Connection = DriverManager.getConnection(myURL,username, password);
}
    catch(SQLException e)
        {
            e.printStackTrace();

            // other exception handling
}
```

Using Character Encoding

Java uses wide characters for strings. If the encoding in the database is not also wide character (e.g., UCS-2), the driver has to know the database code page in order to correctly exchange character data with the database engine. The database character data encoding is specified using the "encoding" attribute in the connection string passed to the driver manager.

Encoding Attribute

The encoding attribute specifies a particular code page to use for translating character data. This can be automated by setting the encoding attribute to "auto." This directs the driver to automatically use the code page used in the database. You can also specify a specific code page. If the encoding attribute is absent, the default operating system code page for the client machine is used. The assumption is that the client and server use the same operating system encoding.

Setting the encoding attribute to "auto" also results in SQL query text being sent to the engine using UTF-8 encoding instead of using the database code page encoding. This will preserve NCHAR string literals in query text.

Example of Using Character Encoding

```
public static void main(String[] args)
{
    //specify latin 2 encoding
    String url = "jdbc:pervasive://MYSERVR:1583/SWEDISH_DB;encoding=cp850";
        try{
        Class.forName("com.pervasive.jdbc.v2.Driver");
        Connection conn = DriverManager.getConnection(url);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select * from SwedishTable");
        rs.close();
        stmt.close();
        conn.close();
```

```
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

Notes on Character Encoding

The Zen JDBC driver uses Java native support for code pages. The list of supported code pages can be obtained from the Oracle Corporation website.

Developing Web-based Applications

This section describes how to create web-based applications with the Zen JDBC driver.

Applets

To develop web based applications using JDBC, you need to place the JDBC jar file in the codebase directory containing the applet classes.

For example, if you are developing an application called MyFirstJDBCapplet, you need to place the `pvjdbc2.jar` file (or the Zen jdbc package) in the directory containing the MyFirstJDBCapplet class. For example, it might be `C:\inetpub\wwwroot\myjdbc\`. This enables the client web browser to be able to download the JDBC driver over the network and connect to the database.

Also, if you use the JAR file, you need to put the archive parameter within the `<APPLET>` tag. For example,

```
<applet CODE="MyFirstJDBCapplet.class" ARCHIVE="pvjdbc2.jar" WIDTH=641 HEIGHT=554>
```

Note: The Zen engine must be running on the web server that hosts the applet.

Servlets and Java Server Pages

JSP can be used to create web-based applications with the Zen JDBC driver.

The following is a sample Java server page for displaying one table in the Demodata sample database included with Zen:

```
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>

 <%
 Class.forName("com.pervasive.jdbc.v2.Driver");
 Connection con = DriverManager.getConnection("jdbc:pervasive://localhost:1583/demodata");
 PreparedStatement stmt = con.prepareStatement("SELECT * FROM Course ORDER BY Name");
 ResultSet rs = stmt.executeQuery();
 %>

 <html>
 <head>
 <title>JSP Sample</title>
 </head>
 <body>

<h1>JSP Sample</h1>
<h2>Course table in Demodata database</h2>
<p>
This example opens the Course table from the Demodata
database and displays the contents of the table.
</p>
```

```
    <table border=1 cellpadding=5>
    <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Credit Hours</th>
    <th>Department Name</th>
    </tr>

    <% while(rs.next()) { %>
           <tr>
           <td><%= rs.getString("Name") %></td>
           <td><%= rs.getString("Description") %></td>
           <td><%= rs.getString("Credit_Hours") %></td>
           <td><%= rs.getString("Dept_Name") %></td>
           </tr>
   <% } %>

</table>

</body>
</html>
```

Information on Servlets and JSP

For more information about servlets and JSP, see the Oracle website.

JDBC 2.0 Standard Extension API

Because connection strings are vendor-specific, Java specifies a DataSource interface. It takes advantage of JNDI, which functions as a Java registry. The DataSource interface allows JDBC developers to create named databases. As a developer, you register the database in JNDI along with the vendor-specific driver information. Then, your JDBC applications can be completely database agnostic and be "pure" JDBC.

The Zen JDBC driver supports the JDBC 2.0 Standard Extension API. Currently, the Zen JDBC driver supports the following interfaces:

- javax.sql.ConnectionEvent
- javax.sql.ConnectionEventListener
- javax.sql.ConnectionPoolDataSource
- javax.sql.DataSource
- javax.sql.PooledConnection

Note: These interfaces are packaged separately in pvjdbc2x.jar in order to keep the core JDBC API 100% Java.

Although at this time Zen does not provide implementation of RowSet interfaces, Zen JDBC driver has been tested with Oracle's implementation of RowSet interface.

DataSource

Java provides a way for application developers to write applications that are driver independent. By using the DataSource interface and JNDI, applications can access data using standard methods and eliminate driver specific elements such as connection strings. In order to use DataSource interface, a database has to be registered with a JNDI service provider. An application can then access it by name.

The following is an example of using the DataSource interface:

```
// This code will have to be executed by the administrator in order to register the
// DataSource. This sample uses Oaracle's reference JNDI implementation.

public void registerDataSources()
    {
// this example uses the JNDI file system
// object as its registry

     Context ctx;
     jndiDir = "c:\\jndi";

    try
```

```
    {
        Hashtable env = new Hashtable (5);
        env.put (Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");

        env.put(Context.PROVIDER_URL, jndiDir);
        ctx = new InitialContext(env);
        }
        catch (Exception e)
        {
            System.out.println(e.toString());
        }

        //register demodata as regular data source
    com.pervasive.jdbc.v2.DataSource ds = new com.pervasive.jdbc.v2.DataSource();
        String dsName = "";

        try
        {
            // Set the user name, password, driver type and network protocol
            ds.setUser("administrator");
            ds.setPassword("admin");
            ds.setPortNumber("1583");
            ds.setDatabaseName("DEMODATA");
            ds.setServerName("127.0.0.1");
            ds.setDataSourceName("DEMODATA_DATA_SOURCE");
            ds.setEncoding("cp850");
            dsName = "jdbc/demodata";

            // Bind it
            try
            {
                ctx.bind(dsName,ds);
                    System.out.println("Bound data source [" + dsName + "]");
            }
            catch (NameAlreadyBoundException ne)
            {
                System.out.println("Data source [" + dsName + "] already bound");
            }
            catch (Throwable e)
                {
                System.out.println("Error in JNDI binding occurred:");
                throw new Exception(e.toString());
            }
        }
    }
}

//In order to use this DataSource in the application, the following code needs to be executed.

public DataSource lookupDataSource(String ln) throws SQLException
    {
      Object ods = null;
      Context ctx;

        try
        {
            Hashtable env = new Hashtable (5);
            env.put (Context.INITIAL_CONTEXT_FACTORY,
                "com.sun.jndi.fscontext.RefFSContextFactory");

// This will create the jndi directory and return its name
// if the directory does not already exist.

String jndiDir = "c:\\jndi";
```

```
            env.put(Context.PROVIDER_URL, jndiDir);
            ctx = new InitialContext(env);
        }
        catch (Exception e)
        {
            System.out.println(e.toString());
        }
        try
        {
            ods = ctx.lookup(ln);
            if (ods != null)
                System.out.println("Found data source [" + ln + "]");
            else
                System.out.println("Could not find data source [" + ln + "]");
        }
        catch (Exception e)
        {
            throw new SQLException(e.toString());
        }

        return (DataSource)ods;
    }

// Note that ConnectionPoolDataSource is handled similarly.
```

Connection and Concurrency

A single Zen JDBC connection can easily serve multiple threads. However, while the Connection may be thread-safe, the objects created by the Connection are not. For example, a user can create four threads. Each of these threads could be given their own **Statement** object (all created by the same **Connection** object). All four threads could be sending or requesting data over the same connection at the same time. This works because all four **Statement** objects have a reference to the same **Connection** object and their reading and writing is synchronized on this object. However, thread #1 cannot access the Statement object in thread #2 without this access being synchronized. The above is true for all other objects in the JDBC API.

Scrollable Result Sets

Scrollable result sets allow you to move forward and backward through a result set. This type of movement is classified as either relative or absolute. You can position absolutely on any scrollable result set by calling the methods `first()`, `last()`, `beforeFirst()`, `afterLast()`, and `absolute()`. Relative positioning is done with the methods `next()`, `previous()`, and `relative()`.

A scrollable result set can also either be updateable or read-only. This refers to whether or not you are able to make changes to the underlying database. Another term, sensitivity, refers to whether these changes are reflected in your current result set.

A sensitive result set will reflect any insert, updates, or deletes made to it. In the case of Zen, an insensitive result set does not reflect any changes made, since it is a static snapshot of the data. In other words, you do not see your updates or those made by anyone else.

Sensitive and insensitive result sets correspond to dynamic and static in ODBC, respectively. A sensitive result set reflects your own changes and can reflect others changes if the transaction isolation level is set to `READ_COMMITTED`. Transaction isolation is set using the **Connection** object. The result set type is set upon statement creation.

If your result set is insensitive, then it is possible to make calls to the method `getRow()` in order to determine your current row number. On an insensitive result set, you can also make calls to `isLast()`, `isFirst()`, `isBeforeFirst()`, and `isAfterLast()`. On a sensitive result set, you can only make calls to `isBeforeFirst()` and `isAfterLast()`. Also, on an insensitive result set, the driver will honor the fetch direction suggested by the user. The driver ignores the suggested fetch direction on sensitive result sets.

JDBC Programming Sample

The following example creates a connection to the database named DB on server MYSERVER. It then creates a statement object on that connection that is sensitive and updateable. Using the statement object a "SELECT" query is performed. Once the result set object is obtained a call to "absolute" is made in order to move to the fifth row. Once on the fifth row the second column is updated with an integer value of 101. Then a call to "updateRow" is made to actually make the update.

```
Class.forName("com.pervasive.jdbc.v2.Driver");
Connection conn=
DriverManager.getConnection("jdbc:pervasive://MYSERVER:1583/DB");

Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs =
m_stmt.executeQuery("SELECT * FROM mytable");

rs.absolute(5);
rs.updateInt(2, 101);
rs.updateRow();

rs.close();
stmt.close();
conn.close();
```

JDBC API Reference

The JDBC API is a standard interface to databases using the Java programming language. The following topics discuss this interface:

- JDBC API Reference
- JDBC Samples

JDBC API Reference

JDBC is a standard API that is documented on the Oracle website. See the JDBC and the JDBC documentation content, noting the API limitations of the driver in Zen JDBC Driver Limitations.

Other useful sites for JDBC programming include Tomcat information at jakarta.apache.org and Apache information at www.apache.org.

For conceptual information on programming with the JDBC driver, see the following topics:

- Introducing the Zen JDBC Driver
- Programming with the Zen JDBC 2 Driver

JDBC Samples

The Zen SDK includes JDBC samples in the samples directory under your Zen SDK installation directory. If you installed to the default location, it is *file_path*\Zen\sdk\samples\jdbc.

For default locations of Zen files, see Where are the files installed? in *Getting Started with Zen*.