Database System

JDBC API

Muhammad Tariq Mahmood
tariq@koreatech.ac.kr
School of Computer Science and Engineering
Korea University of Technology and Education

What are JDBC and JDBC API?

- JDBC is a standard *Java Database Connectivity* and JDBC API can be considered as a *Java Database Connectivity Application Programming Interface* (JDBC API).

- All components and techniques of JDBC are embedded and implemented in JDBC API.

- Basically, the JDBC API is composed of a set of classes and interfaces used to interact with databases from Java applications.

- Generally, the JDBC API perform the following three functions

  - Establish a connection between your Java application and related databases
  - Build and execute SQL statements
  - Process the results

What are JDBC and JDBC API?

- Different database vendors provide various JDBC drivers to support their applications to different databases. The most popular JDBC components are located at the following packages:

- java.sql: contains the standard JDBC components

- javax.sql: contains the Standard Extension of JDBC, which provides additional features such as Java Naming and Directory Interface (JNDI) and Java Transaction Service (JTS).

- oracle.jdbc: contains the extended functions provided by the java.sql and javax.sql interfaces.

- oracle.sql: contains classes and interfaces that provide Java mappings to SQL data types.

What are JDBC and JDBC API?

- Generally, JDBC API enables users to access virtually any kind of tabular data source such as spreadsheets or flat files from a Java application. It also provides connectivity to a wide scope of SQL or Oracle databases.
- One of the most important advantages of using JDBC is that it allows users to access any kind of relational database in a same coding way, which means that the user can develop one program with the same coding to access either a SQL Server database or an Oracle database, or MySQL database without coding modification.
- The JDBC 3.0 and JDBC 4.0 Specifications contain additional features, such as extensions to the support to various data types, MetaData components and improvements on some interfaces.

JDBC Components and Architecture

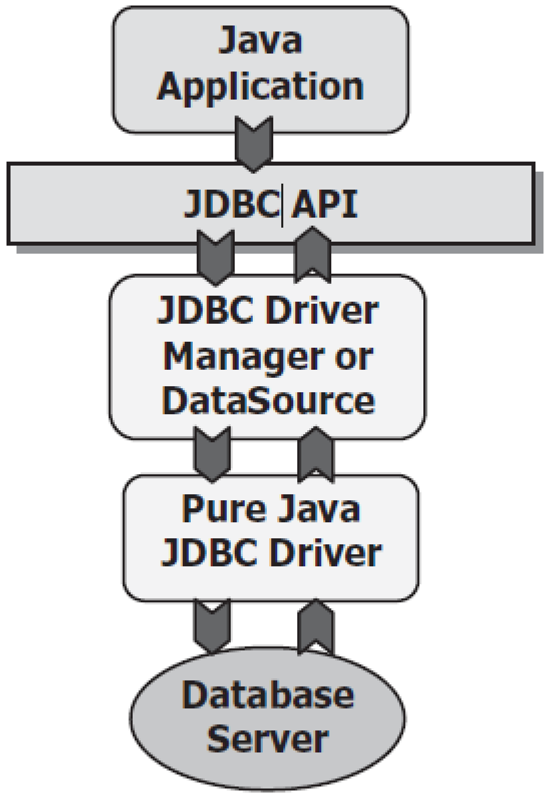
- The core of JDBC API is called a JDBC driver, which implements all JDBC components, including the classes and interfaces, to build a connection and manipulate data between your Java application and selected database.

- Exactly a JDBC driver, which is a class that is composed of a set of methods, builds a connection and accesses databases through those methods.

- The JDBC API contains two major sets of interfaces: the first is the JDBC API for application writers (interface to your Java applications), and the second is the lower-level JDBC driver API for driver writers (interface to your database).

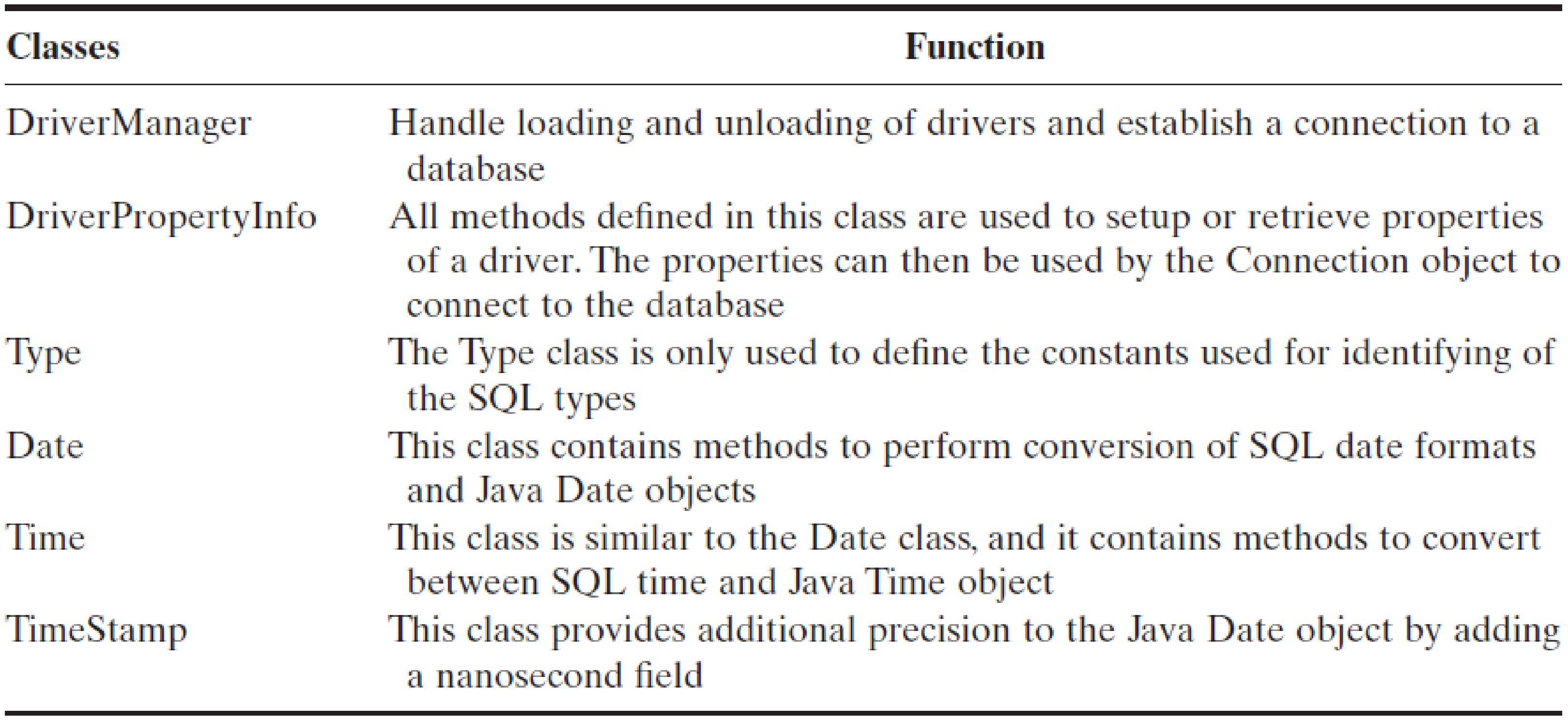
- The JDBC API is composed of a set of classes and interfaces used to interact with databases from Java applications.

JDBC Components and Architecture



The components and architecture of a JDBC API.

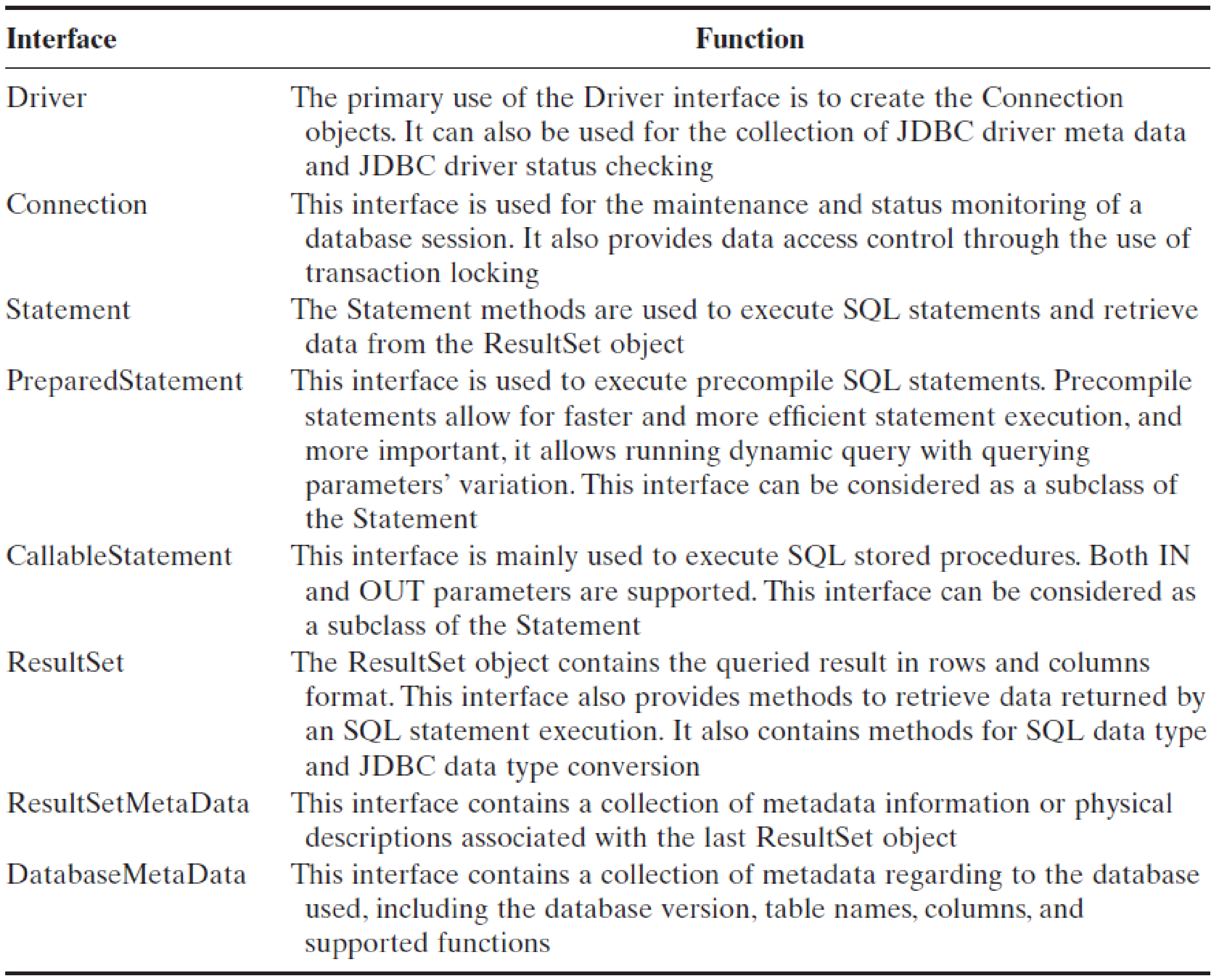
JDBC Components and Architecture



Classes	Function
DriverManager	Handle loading and unloading of drivers and establish a connection to a database
DriverPropertyInfo	All methods defined in this class are used to setup or retrieve properties of a driver. The properties can then be used by the Connection object to connect to the database
Type	The Type class is only used to define the constants used for identifying of the SQL types
Date	This class contains methods to perform conversion of SQL date formats and Java Date objects
Time	This class is similar to the Date class, and it contains methods to convert between SQL time and Java Time object
TimeStamp	This class provides additional precision to the Java Date object by adding a nanosecond field

Classes defined in the JDBC API

JDBC Components and Architecture



Interface	Function
Driver	The primary use of the Driver interface is to create the Connection objects. It can also be used for the collection of JDBC driver meta data and JDBC driver status checking
Connection	This interface is used for the maintenance and status monitoring of a database session. It also provides data access control through the use of transaction locking
Statement	The Statement methods are used to execute SQL statements and retrieve data from the ResultSet object
PreparedStatement	This interface is used to execute precompile SQL statements. Precompile statements allow for faster and more efficient statement execution, and more important, it allows running dynamic query with querying parameters' variation. This interface can be considered as a subclass of the Statement
CallableStatement	This interface is mainly used to execute SQL stored procedures. Both IN and OUT parameters are supported. This interface can be considered as a subclass of the Statement
ResultSet	The ResultSet object contains the queried result in rows and columns format. This interface also provides methods to retrieve data returned by an SQL statement execution. It also contains methods for SQL data type and JDBC data type conversion
ResultSetMetaData	This interface contains a collection of metadata information or physical descriptions associated with the last ResultSet object
DatabaseMetaData	This interface contains a collection of metadata regarding to the database used, including the database version, table names, columns, and supported functions

Interfaces defined in the JDBC API

How Does JDBC Work?

- As we mentioned in the last section, the JDBC API has three functions:

    1. Setup a connection between your Java application and your database;
    2. Build and execute SQL statements
    3. Process results.

- We will discuss these functions in more details in this section based on the JDBC architecture shown in Figure 3.1.

Establish a Connection

- JDBC Driver class contains six method and one of the most important methods is the connect() method, which is used to connect to the database.

- When using this Driver class, a point to be noted is that most methods defined in the Driver class never be called directly, instead, they should be called via the DriverManager class methods.

Using DriverManager to Establish a Connection

- The operational sequence of loading and registering a JDBC driver is:
  1. Call class methods in the DriverManager class to load the driver into the Java interpreter
  2. Register the driver using the registerDriver() method

- To load and register a JDBC driver, two popular methods can be used;

  1. Use Class.forName() method:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

  2. Create a new instance of the Driver class:

```
Driver  sqlDriver = new  com.microsoft.sqlserver.jdbc.SQLServerDriver;
```

Using DataSource to Establish a Connection

- A DataSource object is normally registered with a Java Naming and Directory Interface (JNDI) naming service.

- This means that an application can retrieve a DataSource object by name from the naming service independently of the system configuration.

- Perform the following three operations to deploy a DataSource object:

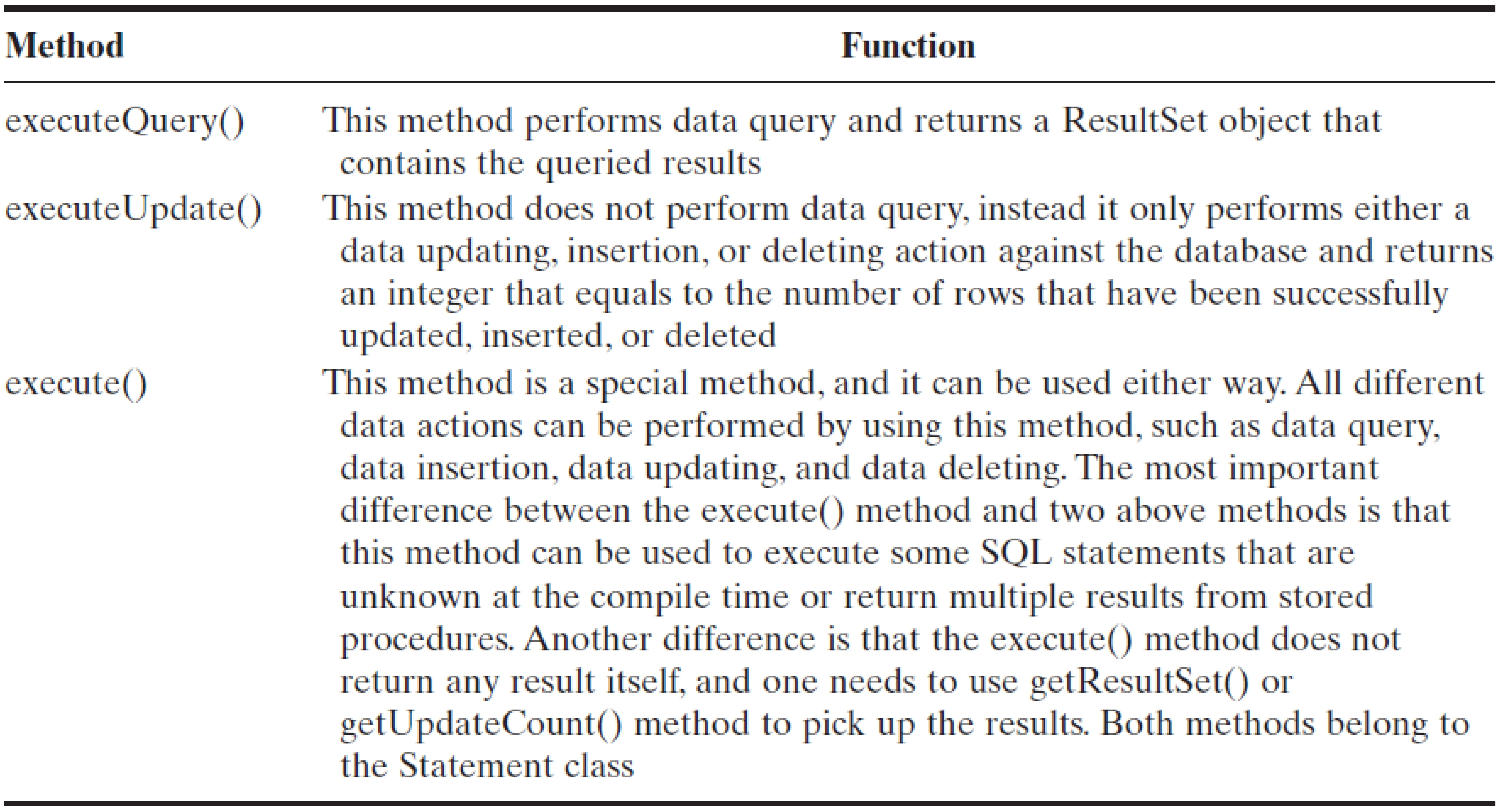
  1. Create an instance of the DataSource class
  2. Set its properties using setter methods
  3. Register it with a JNDI naming service

- The getConnection() method is always used to setup this connection.

Build and Execute SQL Statements

- To build a SQL statement, one needs to call the method createStatement() that belongs to the Connection class to create a new Statement object.
- Regularly, there are three type of Statement objects widely implemented in the JDBC API; Statement, PreparedStatement and CallableStatement.
- The relationship among these three classes is: the PreparedStatement and CallableStatement classes are the subclasses of the Statement class.
- To execute a SQL statement, one of the following three methods can be called:
  1. executeQuery()
  2. executeUpdate()
  3. execute()

Process the Results



Method	Function
executeQuery()	This method performs data query and returns a ResultSet object that contains the queried results
executeUpdate()	This method does not perform data query, instead it only performs either a data updating, insertion, or deleting action against the database and returns an integer that equals to the number of rows that have been successfully updated, inserted, or deleted
execute()	This method is a special method, and it can be used either way. All different data actions can be performed by using this method, such as data query, data insertion, data updating, and data deleting. The most important difference between the execute() method and two above methods is that this method can be used to execute some SQL statements that are unknown at the compile time or return multiple results from stored procedures. Another difference is that the execute() method does not return any result itself, and one needs to use getResultSet() or getUpdateCount() method to pick up the results. Both methods belong to the Statement class

The function of three SQL statements execution methods

JDBC Driver and Driver Types

- The JDBC driver builds a bridge between your Java applications and your desired database, and works as an intermediate-level translator to perform a double-direction conversion:
  1. convert your high-level Java codes to the low-level native codes to interface to the database,
  2. convert the low-level native commands from the database to your high-level Java codes.
- Generally, the JDBC API will not contain any JDBC driver and you need to download a desired JDBC driver from the corresponding vendor if you want to use a specified driver.
- Based on the different configurations, JDBC drivers can be categorized into the following four types.

Type I: JDBC-ODBC Bridge Driver

- Open Database Connectivity (ODBC) is a Microsoft-based database Application Programming Interface (API) and it aimed to make it independent of programming languages, database systems, and operating systems.

- In other words, the ODBC is a database and operating system independent API and it can access any database in any platform without problem at all.

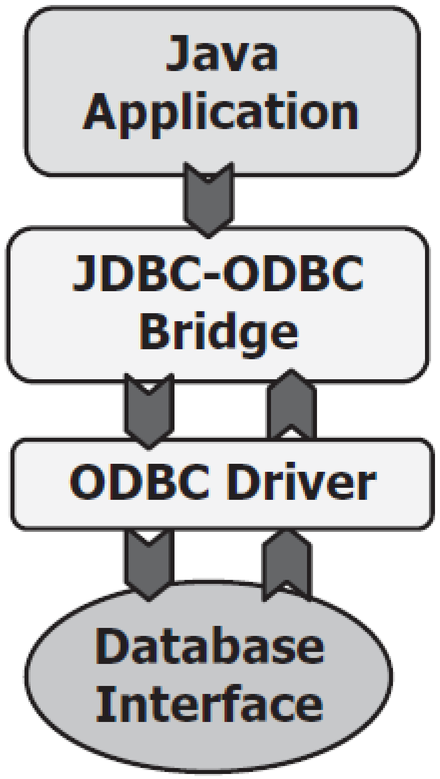
- Basically, ODBC is built and based on various Call Level Interface (CLI) specifications from the SQL Access Group and X/Open techniques.

- To access an ODBC to interface to a desired database, a JDBC-ODBC Bridge works like a translator or a converter, which interpreters the JDBC requests to the CLI in ODBC when a request is sent from the JDBC to the ODBC, and perform an inverse translation (from CLI in ODBC to JDBC) when a result is returned from the database.

Type I: JDBC–ODBC Bridge Driver

- The advantage of using Type I driver is simplicity since we do not need to know the details inside ODBC and transactions between the ODBC and DBMS.

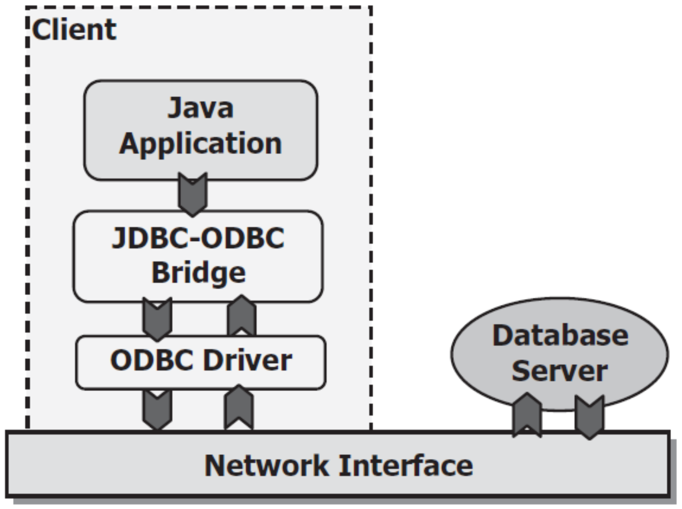
- it is a typical Java standard–alone application that uses JDBC–ODBC Bridge Driver to access a local database, and it will work fine.



Type I: JDBC–ODBC Bridge Driver

- Figure shows a Java based 2–tier application.

- The problem is that the network standard security manager will not allow the ODBC that is downloaded as an applet to access any local files when you build a Java Applet application to access a database located in a database server.



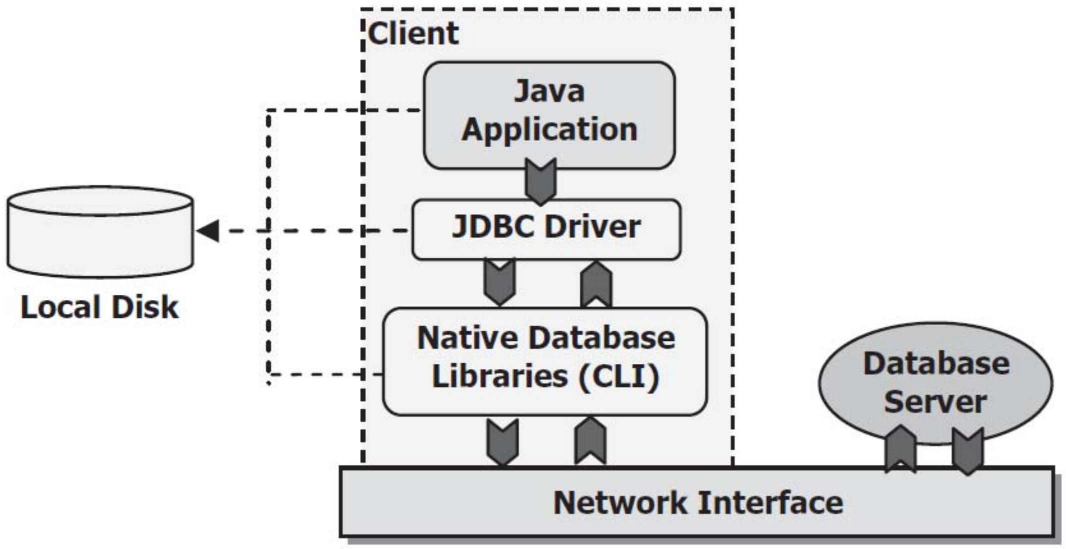
- Therefore, it is impossible to build a Java Applet application with this JDBC–ODBC Bridge Driver configuration.

Type II: Native-API-Partly-Java Driver

- The Native-API-Partly-Java driver makes use of local native libraries to communicate with the database.
- The driver does this by making calls to the locally installed native call level interface (CLI) using a native language, either C or C++, to access the database.
- The CLI libraries are responsible for the actual communications with the database server. When a client application makes a database accessing request, the driver translates the JDBC request to the native method call and passes the request to the native CLI.
- After the database processed the request, results will be translated from their native language back to the JDBC and presented to the client application.

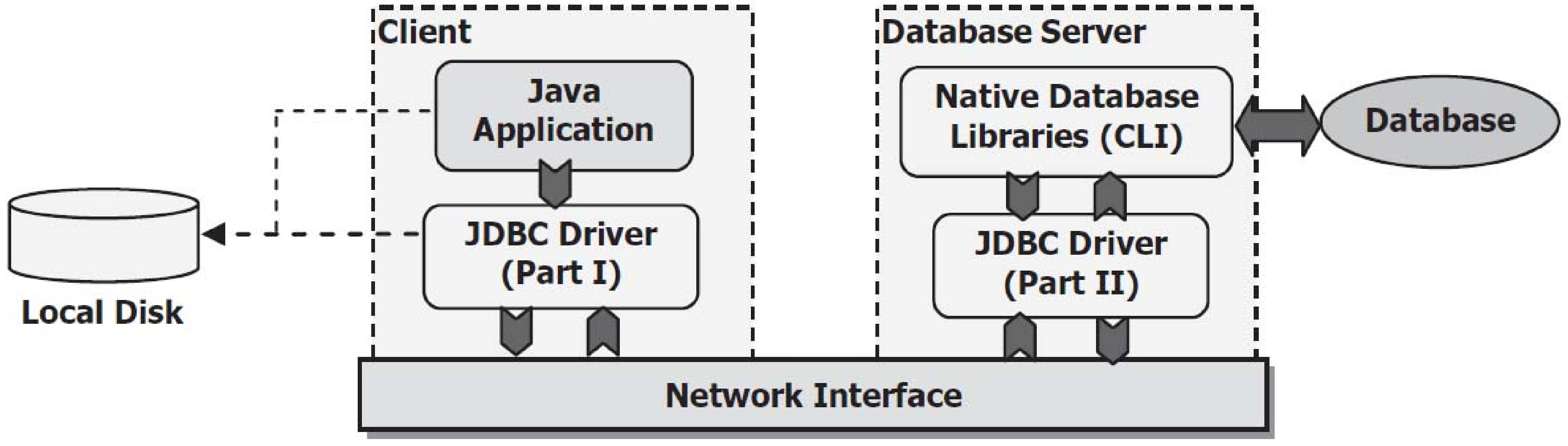
3.4.2 Type II: Native-API-Partly-Java Driver

- Compared with Type I driver, the communications between the driver and the database are performed by using the native CLI without any translation between JDBC and ODBC driver.
- The speed and efficiency of Type II driver is higher than that of Type I driver.
- When available, Type II drivers are recommended over Type I drivers.



Type III: JDBC-Net-All-Java Driver

- Basically the Type III drivers are similar with Type II drivers and the only difference between them is the replacement of the native database access libraries.



Type III: JDBC-Net-All-Java Driver

- For both Type I and Type II drivers, either the ODBC driver or the native CLI libraries must be installed and located on the client machine.
- All communications between the server processes and the JDBC driver have been through native program interface.
- However, in Type III driver, the native CLI libraries are placed on a server and the driver uses a network protocol to facilitate communications between the application and the driver.
- The result of this modification is to separate the driver into two parts:
  1) a part of JDBC driver that is an all-Java portion can be downloaded to the client and
  2) a server portion containing both another part of JDBC driver and native CLI methods.
- All communications between the application and the database server are 100% Java to Java. However, the communication between the database and the server is still done via a native database CLI.

Type III: JDBC-Net-All-Java Driver

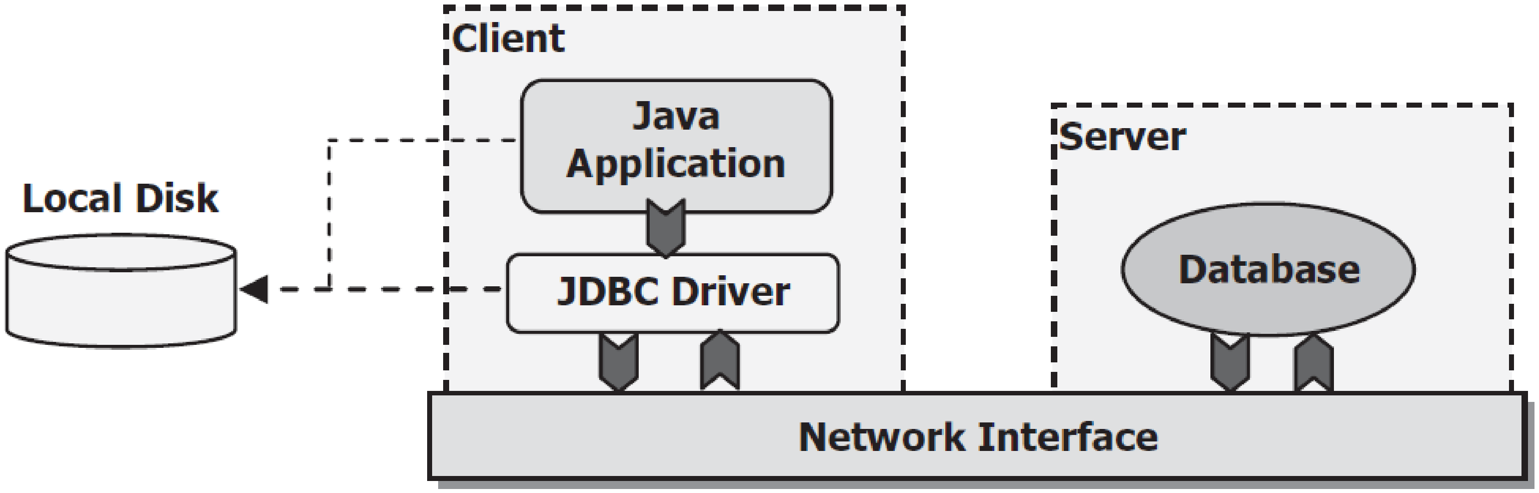
- It can be found that the client does not need to perform either database-specified protocol translation or a Java-to-CLI translation by using Type III drivers, and this will greatly reduce the working loads for the client machine and the client piece of a Type III driver only needs to translate requests into the network protocol to communicate with the database server.

- Another advantage of using a Type III driver is that the second part of the Type III driver, which is used to communicate with the database native libraries, does not need to be downloaded to the client, and as a result of this fact, Type III drivers are not subject to the same security restrictions found as Types I and II did.

- Since all database-related codes reside on the server side, a large driver that is capable of connecting to many different databases can be built.

Type IV: Native–Protocol–All–Java Driver

- Type IV drivers are capable of communicating directly with the database without the need for any type of translation since they are 100% Java without using any CLI native libraries.
- Figure shows a typical Type IV driver configuration.



Type IV: Native-Protocol-All-Java Driver

- The key issue of using a Type IV driver is that the native database protocol will be rewritten to converts the JDBC calls into vendor specific protocol calls, and the result of this rewritten is that the driver can directly interact with the database without needing any other translations.
- Therefore, Type IV drivers are the fastest drivers compared with all other three-type drivers, Types I ~ III.
- By using a Type IV driver, it will greatly simplify database access for applets by eliminating the need for native CLI libraries.

JDBC Standard Extension API

- Besides the standard JDBC API (or core API), Sun added an extension package called JDBC 2.0 Standard Extension API to support extended database operations.
- This package contains the following components:
  - JDBC DataSource
  - JDBC driver-based connection pooling
  - JDBC RowSet
  - Distributed transactions

JDBC Application Design Considerations

- JDBC API supports both two-tier and three-tier models for database accesses.
- In a two-tier model, a Java application or an applet can communicate directly with the database.
- In a three-tier model, commands are sent to a middle-tier, which sends the messages to the database. In return, the result of the database query is sent to the middle-tier that finally directs it to the application or applet. The presence of a middle-tier has a number of advantages, such as a tight control over changes done to the database.

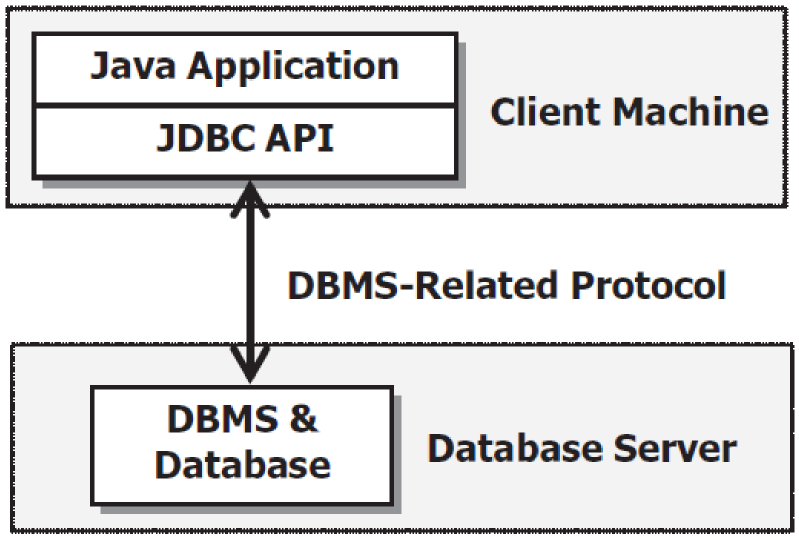
Two-Tier Client-Server Model

- In a two-tier client-server model, a Java application can directly communicate with the database. In fact, the so-called two-tier model means that the Java application and the target database can be installed in two components with two layers:
  - Application layer, which includes the JDBC driver, user interface and the whole Java application, installed in a client machine.
  - Database layer, which includes the RDBMS and the database, installed in a database server.
- A client-server configuration is a special case of the two-tier model, where the database is located on another machine called the database server.
- The Java application program runs on the client machine that is connected to the database server through a network.

Two-Tier Client-Server Model

- It can be found from Figure that both Java application and JDBC API are located at the first layer, or the client machine and the DBMS and database are located at the second layer or the database server.

- A DBMS-Related protocol is used as a tool to communicate between these two layers.



Three-Tier Client-Server Model

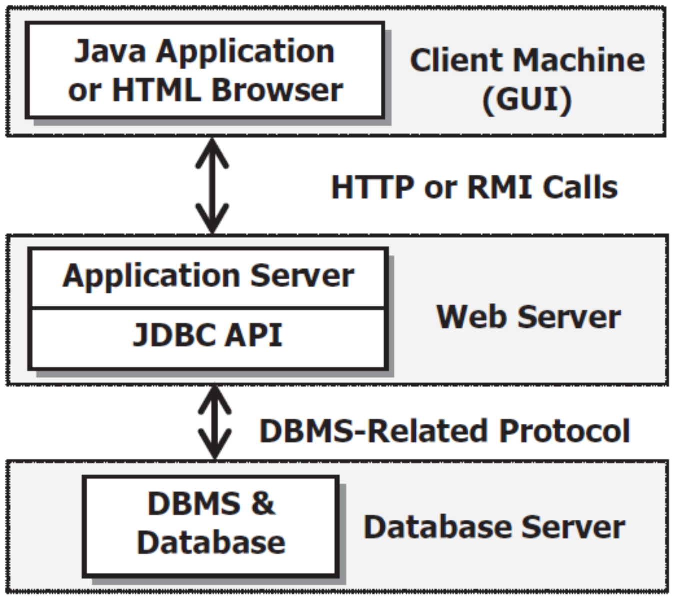
This model can be represented by the following three layers:

- Client layer, which includes a Web browser with some language-specified virtual machines, installed in a client machine.
- Application server layer, which includes Java Web applications or Java Web services, installed in a Web server.
  - This layer is used to handle the business logic or application logic. This may be implemented using Java Servlet engines, Java Server Pages or Java Server Faces. The JDBC driver is also located in this layer.
- Database layer, which includes the RDBMS and the database, installed in a database server.

Three-Tier Client-Server Model

Advantages of using a three-tier configuration:

- Application performance can be greatly improved by separating the application server and database server.
- Business logic is clearly separated from the database.
- Client application can then use a simple protocol to access the server.



JDBC Applications Fundamentals

- To run a Java database application to perform data actions against the database, a JDBC API needs to perform the following operations:
  - Establish a connection between your Java application and related databases
  - Build and execute SQL statements
  - Process the results
- The above three steps can be divided into following seven steps:
  - Import necessary Java packages, such as java.awt, java.sql and javax.sql
  - Load and register the JDBC driver
  - Establish a connection to the database server
  - Create a SQL statement
  - Execute the built statement
  - Retrieve the executing results
  - Close the statement and connection objects

The DriverManager and Driver Classes

- Four methods in the DriverManager class are widely applied in most database applications:

  getConnection()
  getDriver()
  registerDriver()
  deregisterDriver()

- Note that the getConnection() method has two more overloading methods with different arguments.
- Most popular methods in the Driver class are acceptsURL() and connect().
- Most methods defined in the Driver class will not be called directly in most Java database applications, instead, they will be called indirectly by using the DriverManager class.

DriverManager class

Table 4.1. Methods defined in the DriverManager class

Method	**Function**
deregisterDriver(Driver dr)	Remove a Driver from the driver list
getConnection(String url, Properties login)	Attempt to establish a connection to the referenced database
getConnection(String url, String user, String pswd)	Attempt to establish a connection to the referenced database
getConnection(String url)	Attempt to establish a connection to the referenced database
getDriver(String url)	Locate an appropriate driver for the referenced URL from the driver list
getDrivers()	Get a list of all drivers currently loaded and registered
getLoginTimeout()	Get the maximum time (in seconds) a driver will wait for a connection
getLogStream()	Get the current PrintStream being used by the DriverManager.
Println(String msg)	Print a message to the current LogStream.
registerDriver(Driver dr)	Add the driver to the driver list. This is normally done automatically when the driver is instantiated
setLoginTimeout(int seconds)	Set the maximum time (in seconds) that a driver can wait when attempting to connect to a database before giving up
setLogStream(PrintStream out)	Set the PrintStream to direct logging message to

Driver class

Table 4.2. Methods defined in the Driver class

Method	Function
acceptsURL(String url)	Return a true if the driver is able to open a connection to the database given by the URL
connect(String url, Properties login)	Check the syntax of the URL and the matched drivers in the driver list. Attempt to make a database connection to the given URL
getMajorVersion()	Determine the minor revision number of the driver
getMinorVersion()	Determine the major revision number of the driver
getPropertyInfo(String url, Properties login)	Return an array of DriverPropertyInfo objects describing login properties accepted by the database
jdbcCompliant()	Determine if the driver is JDBC COMPLIANT

Loading and Registering Drivers

- The DriverManager class is a set of utility functions that work with the Driver methods together and manage multiple JDBC drivers by keeping them as a list of drivers loaded. Although loading a driver and registering a driver are two steps, only one method call is necessary to perform these two operations. The operational sequence of loading and registering a JDBC driver is:
  - Call class methods in the DriverManager class to load the driver into the Java interpreter
  - Register the driver using the registerDriver() method

- When loaded, the driver will execute the DriverManager.registerDriver() method to register itself. The above two operations will never be performed until a method in the DriverManager is executed, which means that even both operations have been coded in an application, the driver cannot be loaded and registered until a method such as connect() is first executed.

Loading and Registering Drivers

- To load and register a JDBC driver, two popular methods can be used;

  Use Class.forName() method:
  - Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

  Create a new instance of the Driver class:
  - Driver sqlDriver = new com.microsoft.sqlserver.jdbc.SQLServerDriver;

- Relatively speaking, the first method is more professional since the driver is both loaded and registered when a valid method in the DriverManager class is executed.

- The second method cannot guarantee that the driver has been registered by using the DriverManager.

Getting Connected

- To establish a connection to a database, two methods can be used:
  - Using DriverManager.getConnection() method
  - Using Driver.connect() method
- Before we can take a closer look at these two methods, first let’s have a quick review for all methods defined in these two classes, DriverManager and Driver.

Using the DriverManager.getConnection() Method

- When using the first method DriverManager.getConnection() to establish a database connection, it does not immediately try to do this connection, instead, in order to make this connection more robust, it performs a two-step process.

- The getConnection() method first checks the driver and Uniform Resource Locator (URL) by running a method called acceptsURL() via DriverManager class to test the first driver in the driver list, if no matched driver returns, the acceptURL() method will go to test the next driver in the list. This process continues until each driver is tested or until a matched driver is found.

- If a matched driver is found, the Driver.connect() method will be executed to establish this connection. Otherwise, a SQLException is raised.

Using the Driver.connect() Method

- The Driver.connect() method enable you to create a actual connection to the desired database and returns an associated Connection object. This method accepts the database URL string and a Properties object as its argument.
- A URL indicates the protocol and location of a data source while the properties object normally contains the user login information.
- One point to be noted is that the only time you can use this Driver.connect() method directly is that you have created a new instance of the Driver class.
- A null will be returned if an exception is occurred when this Driver.connect() method is executed, which means that something wrong during this connection operation.

The JDBC Connection URL

- The JDBC URL provides all information for applications to access to a special resource, such as a database.
- A URL contains three parts or three segments;

  protocol name
  sub-protocol
  subname

- Each of these three segments has different function when they worked together to provide unique information for the target database.
- The syntax for a JDBC URL can be presented as:
  - protocol:sub-protocol:subname

The JDBC Connection URL

- The protocol name works as an identifier or indicator to show what kind of protocol should be adopted when connect to a database. For a JDBC driver, the name of the protocol should be jdbc. The protocol name is used to indicate what kind of items to be delivered or connected.

- The sub-protocol is generally used to indicate the type of the database or data source to be connected, such as sqlserver or oracle.

- The subname is used to indicate the address to which the item supposed to be delivered or the location of the database is resided. Generally a subname contains the following information for an address of a resource:
  - Network host name/IP address
  - The database server name
  - The port number
  - The name of the database

The JDBC Connection URL

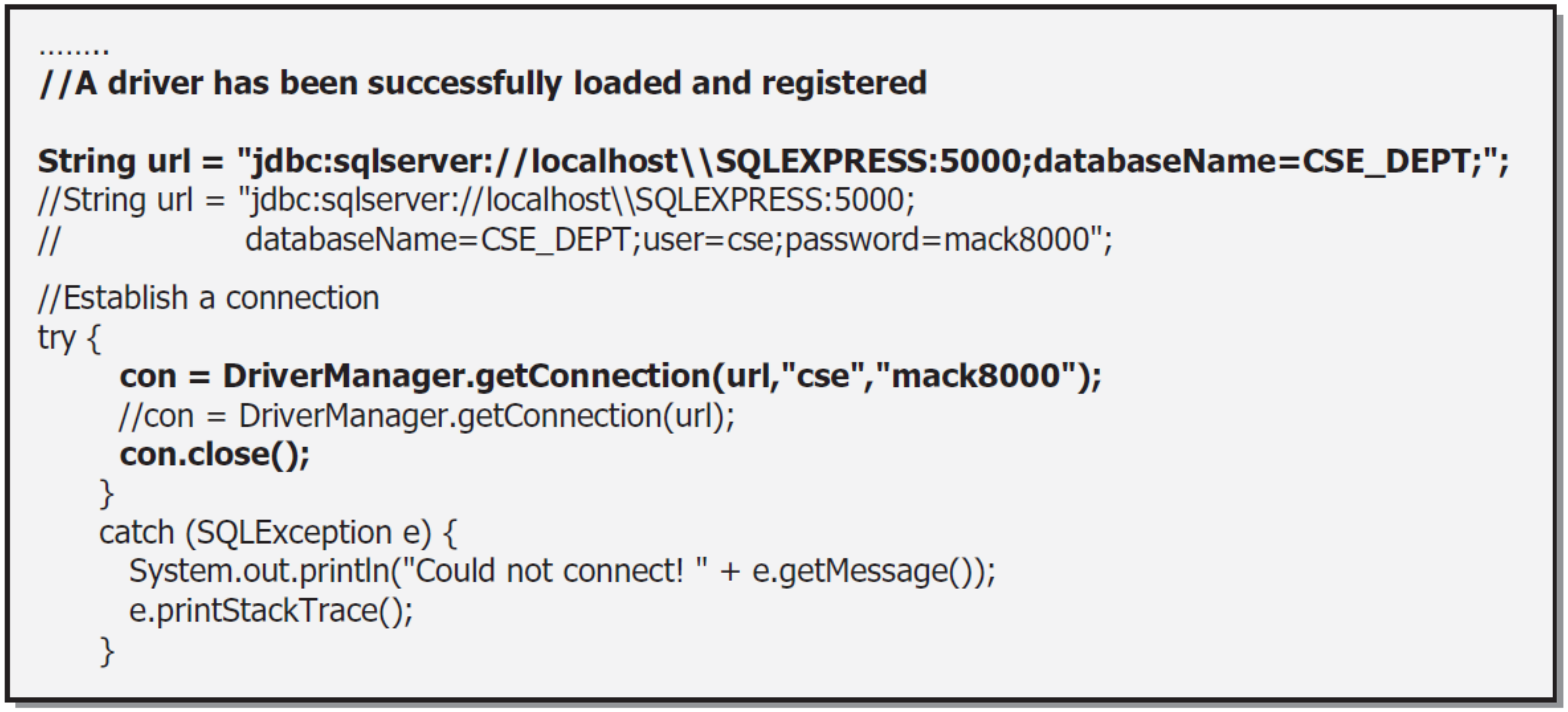
- An example of a subname for our SQL Server database is:
  - localhost\\SQLEXPRESS:5000
- The network host name is localhost, and the server name is SQLEXPRESS and the port number the server used is 5000. You need to use a double slash, either forward or back, to represent a normal slash in this URL string since this is a DOS style string.
- By combining all three segments together, we can get a full JDBC URL. An example URL that is using a SQL Server JDBC driver is:
  - jdbc:sqlserver//localhost\\SQLEXPRESS:5000
- The database's name works as an attribute of the connected database.

Establish a Database Connection

- To establish a database connection, a valid JDBC URL is defined in the first coding line with the following components:
  - The protocol name jdbc
  - The sub-protocol sqlserver
  - The subname localhost\\SQLEXPRESS:5000
  - The database name CSE_DEPT
- Then a try…catch block should be used to try to establish a connection using the getConnection() method with three arguments; URL, username and password.
- After a valid connection is established, a Connection object is returned.

Establish a Database Connection

- Figure shows a piece of example codes to establish a connection using the DriverManager.getConnection() method.
- A valid driver has been loaded and registered before this connection can be established.



```
........
//A driver has been successfully loaded and registered

String url = "jdbc:sqlserver://localhost\\SQLEXPRESS:5000;databaseName=CSE_DEPT;";
//String url = "jdbc:sqlserver://localhost\\SQLEXPRESS:5000;
//                databaseName=CSE_DEPT;user=cse;password=mack8000";

//Establish a connection
try {
        con = DriverManager.getConnection(url,"cse","mack8000");
        //con = DriverManager.getConnection(url);
        con.close();
      }
      catch (SQLException e) {
        System.out.println("Could not connect! " + e.getMessage());
        e.printStackTrace();
      }
```

Figure 4.4. An example coding for the database connection.