کار با بانک اطلاعاتی

امروزه بیشتر نرم‌افزارهای موجود در قالب معماری سرویس دهنده/سرویس گیرنده می‌باشند. برنامه‌های جاوا نیز قادر به ساخت برنامه‌های کاربردی مدل سرویس‌دهنده/سرویس گیرنده و مستقل از پلاتفرم به همراه اتصال به منابع اطلاعاتی از جمله بانک‌اطلاعاتی است. این امر با استفاده از JDBC امکان‌پذیر است. JDBC نیز مستقل از محیط طراحی شده است، بنابراین به هنگام برنامه‌نویسی هیچ نگرانی در مورد تنوع و تفاوتهای بانک های اطلاعاتی وجود ندارد.

JDBC چیست

JDBC یک علامت تجاری و مخفف Java Database Connectivity بوده و به طور خلاصه یک مجموعه کلاس و رابط جاوا برای اجرای دستورات SQL است. این API امکان دسترسی به بانک های اطلاعاتی را فراهم می سازد.

با استفاده از JDBC ارسال دستورات SQL به بانک‌های رابطه‌ای امکان‌پذیر است. به عبارت دیگر با وجود JDBC نیازی به نوشتن برنامه‌های جداگانه برای دسترسی به بانک‌های اطلاعاتی Sybase، Oracle، Informix و DB2 نیست، بلکه با نوشتن یک برنامه و با استفاده از JDBC API می‌توان دستورات SQL را به بانک اطلاعاتی مورد نظر ارسال و اجرا نمود. بدین ترتیب نباید

نگران تفاوت بانک‌های اطلاعاتی و بستر آنها بود. تلفیق جاوا و JDBC به برنامه‌نویس اجازه می‌دهد، یک بار برنامه را نوشته و همه جا اجرا نماید.

JDBC قابلیت‌های جاوا را گسترش داده و با استفاده از آن می‌توان در اپلت‌ها محتویات و اطلاعات جداول بانک اطلاعاتی را نمایش داد یا می‌توان از طریق صفحات وب طیف گسترده‌ای از کامپیوترهای مختلف با سیستم‌عامل‌های ویندوز، مکینتاش و یونیکس را از طریق شبکه اینترنت به بانک اطلاعاتی متصل کرد.

برنامه‌نویسان سطح بالا تلفیق جاوا و JDBC را ترجیح می‌دهند چرا که این روش اطلاعات را به آسانی و به صرفه در اختیار آنها می‌گذارد ، زمان تولید برنامه‌های جدید را کوتاه‌تر کرده و همچنین نصب و کنترل نسخه‌های برنامه را نیز آسان‌تر کرده‌است. برنامه‌نویس می‌تواند یک برنامه را تولید کرده و آن را در سرویس‌دهنده قرار دهد، بطوریکه همه کاربران بتوانند به آخرین نسخه برنامه به آسانی دسترسی پیدا کنند. تلفیق جاوا و JDBC بهترین روش برای دسترسی به اطلاعات برای کاربران است.

وظایف JDBC

به زبان ساده JDBC کارهای زیر را انجام می‌دهد:

1. برقراری ارتباط با بانک اطلاعاتی
2. ارسال دستورات SQL
3. پردازش نتایج
4. قطع ارتباط با بانک اطلاعاتی

قطعه کد زیر مراحل یک این مراحل را نشان می‌دهد.

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

JDBC یک API سطح پایین و اولیه برای برنامه های میانی و API های سطح بالاتر است. بدین معنی که می نوان با استفاده از JDBC API ، مجموعه های تولید کرد که ارتباط با بانک اطلاعاتی را آسانتر نماید. تولید API سطح بالا برای راحتی و فهم بیشتر کاربر یا استفاده‌کننده است. برای مثال می توان از دو نوع API سطح بالاتر زیر نام برد :

- Embedded SQL در جاوا

چندین شرکت برای پیاده‌سازی این API اقدام کرده‌اند. در JDBC دستورات SQL به صورت یک عبارت به متدهای جاوا فرستاده می‌شود. Embedded SQL به برنامه‌نویس اجازه می‌دهد دستورات SQL را بطور صریح در کد جاوا وارد کند. برای مثال یک متغیر جاوا را می‌توان در یک جمله SQL برای دسترسی به فیلدی از جدول بانک اطلاعاتی قرار داد. پیش کامپایلر ترکیب دستورات SQL و جاوا را به برنامه جاوا و فراخوانی JDBC تبدیل می‌کند.

- نگاشت مستقیم جدول بانک اطلاعاتی به کلاس‌های جاوا

JavaSoft و چند شرکت دیگر طرح پیاده‌سازی این ایده را اعلام کرده‌اند. در این نگاشت "Object/Relational" هر سطر از یک جدول (رکورد) نمونه‌ای از یک کلاس جاوا و هر ستون (فیلد) یک متغییر از آن نمونه است، پس برنامه‌نویس می‌تواند بطور مستقیم با این اشیاء کار کند و فراخوانی SQL برای بازیابی و ذخیره اطلاعات در لایه های پایین ترانجام می‌شود.

محبوبیت JDBC روز به روز افزایش می‌یابد، بسیاری از برنامه‌نویسان روی ابزارهای مبتنی بر JDBC بمنظور آسانتر ساختن برنامه‌ها فعالیت می‌کنند. همچنین برنامه‌نویسان برنامه‌هایی تولید می‌کنند که دسترسی به بانک‌های اطلاعاتی را برای کاربر نهایی آسان می‌سازد. برای مثال یک برنامه ممکن است منویی از فعالیت‌های قابل انجام روی بانک را نمایش دهد، پس از انتخاب هر آیتم، عمل مربوطه انجام شده و برنامه دستورات SQL مرتبط با آن را تولید و اجرا می‌کند. با داشتن این برنامه نیاز نیست کاربر اطلاعاتی راجع به SQL و نحوه ارتباط با بانک داشته باشد.

JDBC و ODBC

تکنولوژی ODBC[1] شرکت میکروسافت یکی از روش های دسترسی به بانک‌های اطلاعاتی مختلف است. ODBC امکان دسترسی به بیشتر بانک‌های اطلاعاتی را در محیط‌های متفاوت فراهم می کند. از ODBC در جاوا نیز می توان استفاده کرد، این کار با کمک واسط JDBC-ODBC امکان‌پذیر است:

- ODBC برای استفاده مستقیم در جاوا مناسب نیست، چراکه از رابطی به زبان C استفاده می‌کند. استفاده از کدهای محلی C برای امنیت و قابلیت حمل جاوا مشکلاتی بوجود می‌آورد.
- تبدیل ODBC API به Java API مطلوب نیست. برای مثال جاوا فاقد اشاره‌گر است و ODBC استفاده زیادی از اشاره‌گر کرده و این باعث بروز خطاهای زیادی می‌شود.
- یادگیری ODBC دشوار است، چراکه خصیصه‌های آسان و پیچیده را با هم ترکیب کرده و برای یک درخواست ساده گزینه‌های پیچیده‌ای ارائه می‌دهد.

مدل‌های دو لایه و سه لایه

JDBC مدل‌های دو و سه لایه را برای دسترسی به بانک‌های اطلاعاتی پشتیبانی می‌کند:

- در مدل دو لایه، یک برنامه یا اپلت جاوا به طور مستقیم با بانک اطلاعاتی صحبت می‌کند. این کار نیازمند برقراری ارتباط مستقیم JDBC با مدیریت بانک اطلاعاتی است. دستورات SQL به بانک اطلاعاتی ارجاع شده و نتایج آن برمی‌گردد. بانک اطلاعاتی ممکن است روی یک ماشین دیگر باشد و کاربر از طریق شبکه به آن متصل شود. این مدل مشابه مدل Client/Server است که در آن ماشین کاربر همان سرویس‌گیرنده و ماشین حاوی بانک اطلاعاتی همان سرویس‌دهنده است، شبکه نیز می‌تواند یک اینترانت یا اینترنت باشد.

- در مدل سه لایه دستورات SQL از طریق یک لایه میانی به بانک اطلاعاتی ارسال می‌شود. بانک اطلاعاتی دستورات SQL را پردازش کرده و نتایج را به لایه میانی برمی‌گرداند و این لایه است که اطلاعات را برای کاربر ارسال می‌کند.

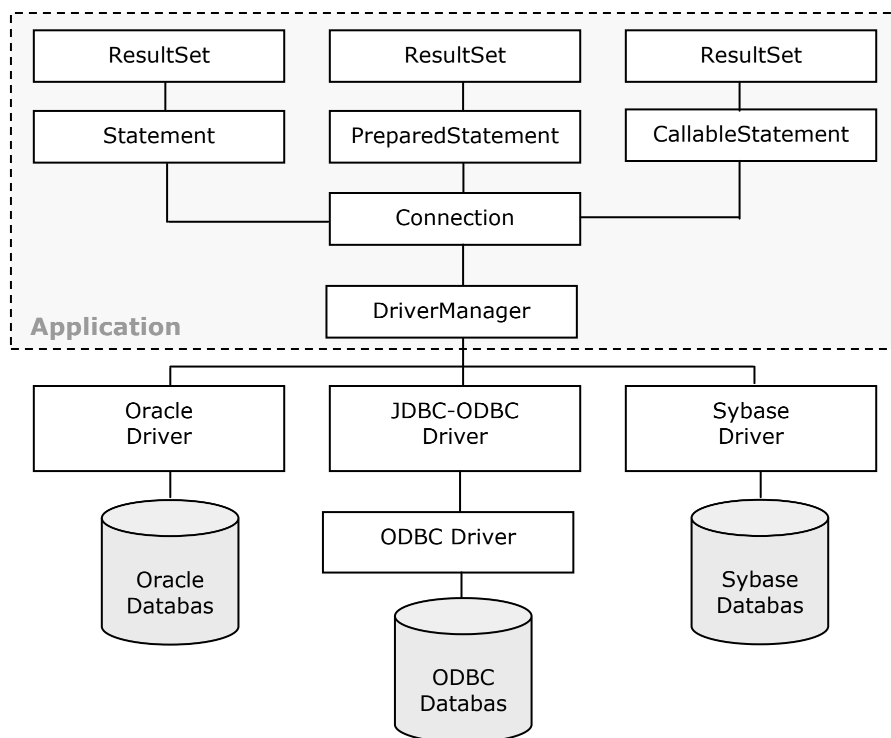
[1] Open Database Connectivity

مدیران MIS مدل سه لایه را ترجیح می‌دهند، چراکه لایه میانی امکان کنترل دسترسی و بروزرسانی اطلاعات را فراهم می‌سازد. ویژگی دیگر آن است که با وجود لایه میانی، کاربر می‌تواند از یک API سطح بالا و آسان استفاده کند. در گذشته لایه‌های میانی با زبان‌های C و C++ نوشته می‌شدند، ولی درحال حاضر با توجه به اینکه بایت کدهای جاوا به راحتی به کد ماشین تبدیل می‌شوند، بهتر است لایه میانی را نیز با جاوا پیاده‌سازی کرد.

معماری JDBC

سیستم‌های بانک اطلاعاتی مختلف دارای ویژگی‌های مشترک زیادی هستند، یکی از این ویژگی‌ها زبان پرس و جو یا SQL است. از طرف دیگر هر بانک اطلاعاتی دارای API منحصر بفرد است که برای برقراری ارتباط باید از آن استفاده کرد. فقط چند API مستقل وجود دارند، ODBC نیز دارای وابستگی‌هایی است که استفاده از آن را محدود می‌کند. JDBC تلاش شرکت سان برای ایجاد یک رابط مستقل برای برقراری ارتباط بین جاوا و بانک‌های اطلاعاتی است. با استفاده از JDBC می‌توان از یک مجموعه استاندارد جهت ارتباط با بانک اطلاعاتی و زبان پرس و جوی SQL بهره گرفت.

JDBC وظایف اصلی بانک اطلاعاتی از قبیل اجرای درخواست، پردازش نتایج و تعریف پیکربندی بانک اطلاعاتی را انجام می‌دهد.



شکل 16-1 ارتباط JDBC و بانک اطلاعاتی

مبانی JDBC

قبل از پرداختن به عناصر JDBC مثال ساده‌ای از مراحل اصلی JDBC بیان می‌شود. مثال زیر ابتدا درایور بانک اطلاعاتی را بارگذاری می‌کند، سپس با بانک اطلاعاتی ارتباط برقرار کرده، چند دستور SQL را اجرا و در نهایت نتایج آن را چاپ می‌کند. در بین انجام کار خطاهای مربوط به بانک اطلاعاتی نیز کنترل می شود.

```
import java.sql.*;

public class JDBCSample {
   public static void main(java.lang.String[] args) {

      try {
         // This is where we load the driver
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (ClassNotFoundException e) {
         System.out.println("Unable to load Driver Class");
         return;
      }

      try {
         // All database access is within a try/catch block. Connect to database,
         // specifying particular database, username, and password
         Connection con = DriverManager.getConnection("jdbc:odbc:companydb", "", "");

         // Create and execute an SQL Statement
         Statement stmt = con.createStatement(  );
         ResultSet rs = stmt.executeQuery("SELECT FIRST_NAME FROM EMPLOYEES");

         // Display the SQL Results
         while(rs.next()) {
            System.out.println(rs.getString("FIRST_NAME"));
         }

         // Make sure our database resources are released
         rs.close();
         stmt.close();
         con.close();
      } catch (SQLException se) {
         // Inform user of any SQL errors
         System.out.println("SQL Exception: " + se.getMessage());
         se.printStackTrace(System.out);
      }
   }
}
```

کد 16-1 مثال ساده JDBC

این مثال با بارگذاری کلاس درایور JDBC (واسط JDBC-ODBC) شروع شده، سپس با استفاده از شیء Connection با بانک اطلاعاتی ارتباط برقرار شده‌است. برای اجرای دستورات SQL از شیء Statement و برای مشاهده نتایج درخواست پرس و جو از شیء ResultSet استفاده شده‌است. سپس نتایج درخواست نمایش داده شده و در نهایت ارتباط با بانک اطلاعاتی قطع و منابع تخصیص یافته آزاد می‌شود و اگر خطایی نیز رخ دهد توسط SQLException کنترل می‌شود. هر برنامه جاوا که از JDBC استفاده کند باید این مراحل را طی کند.

درایورهای JDBC

قبل از استفاده از یک داریور، باید کلاس DriverManager آن ثبت[1] شود. این کار با متد Class.forName() انجام می‌شود.

```
try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Class.forName("com.oracle.jdbc.OracleDriver");
} catch (ClassNotFoundException e) {
   /* Handle Exception */
}
```

دلیل اینکه برنامه‌ها از متد Class.forName() استفاده می‌کنند این است که اطلاعات درایور JDBC می‌تواند دینامیک باشد (برای مثال از یک فایل Properties خوانده شود). راه دیگر برای ثبت درایور JDBC، اضافه‌کردن آن به jdbc.drivers است. برای استفاده از این تکنیک می‌توان عبارت مشابه کد زیر را به ~/.hotjava/properties اضافه کرد (در سیستم‌عامل ویندوز این فایل در دایرکتوری نصب SDK جاوا است).

```
jdbc.drivers=com.oracle.jdbc.OracleDriver:foo.driver.dbDriver:com.al.AlDriver;
jdbc:oracle:thin:@site:port:database
                              jdbc:odbc:datasource[;attribute-name=attribute-value]*
```

[1] Register

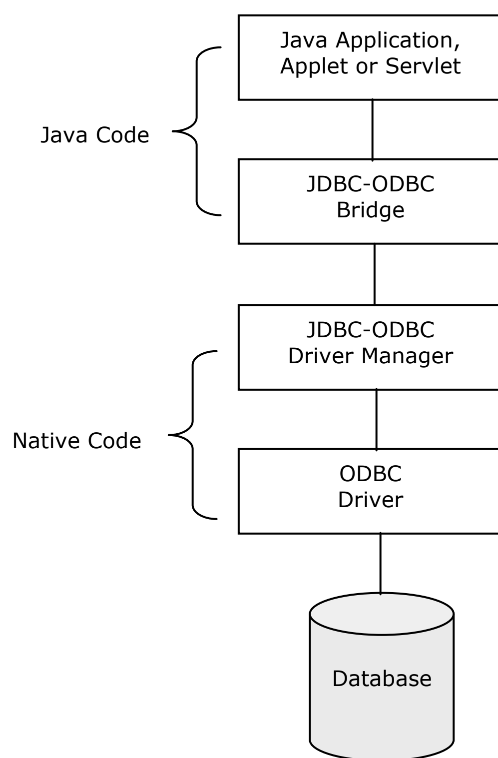
نام درایورها باید با کاراکتر ':' از هم جدا شوند، در انتهای هر خط نیز باید کاراکتر ';' ذکر شود. برنامه‌های جاوا کمتر از این روش استفاده می‌کنند، چرا که نیازمند پیکربندی اضافه در دستگاه کاربر یا سرویس‌گیرنده است. هر کاربر باید درایور JDBC مخصوص بخود را در این فایل داشته باشد.

انواع درایور JDBC

چهار نوع درایور JDBC وجود دارد. آشنایی با نقاط قوت و ضعف هر یک از آنها می‌تواند برای انتخاب داریور مناسب، مفید باشد.

1. نوع یک : واسط JDBC-ODBC

همانطور که بیان شد، واسط JDBC-ODBC توسط شرکت سان به عنوان بخشی از JDK 1.1 به بعد ارائه شد. واسط JDBC-ODBC قسمتی از بسته sun.jdbc.odbc بوده که از متدهای بومی ODBC استفاده می‌کند و به همین دلیل دارای محدودیت‌هایی است.



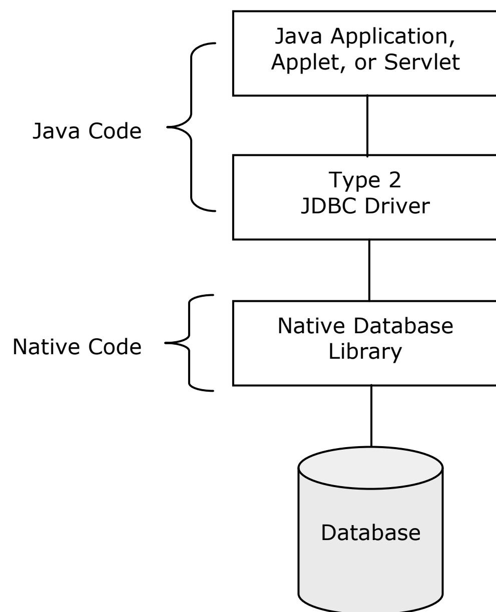
شکل 2-16 واسط JDBC و ODBC

در هر یک از موارد زیر می‌توان از این نوع درایور استفاده کرد:

- □ نمونه‌سازی سریع سیستم.
- □ سیستم‌های بانک اطلاعاتی سه لایه.
- □ سیستم‌های بانک اطلاعاتی که فاقد درایور JDBC هستند.
- □ راه حل‌های ارزانی قیمت.

2. نوع دو : Java to Native API

این درایور از کتابخانه‌های بومی استفاده کرده و بطور مستقیم با بانک اطلاعاتی ارتباط برقرار می‌کند. این نوع درایور همان محدودیت‌های واسط JDBC-ODBC را دارد، چراکه از کدهای بومی استفاده می‌کند. بیشترین محدودیت این درایور آن است که نمی‌توان از آن در اپلت‌های نا امن استفاده کرد. از آنجاییکه این درایور از کدهای بومی استفاده می‌کند، نصب این کدها روی دستگاهی که از این درایور استفاده می‌کند الزامی است. بیشتر عرضه کنندگان بانک اطلاعاتی داریور نوع دو را به همراه محصولاتشان ارائه می‌دهند.



شکل 3-16 درایور بومی جاوا

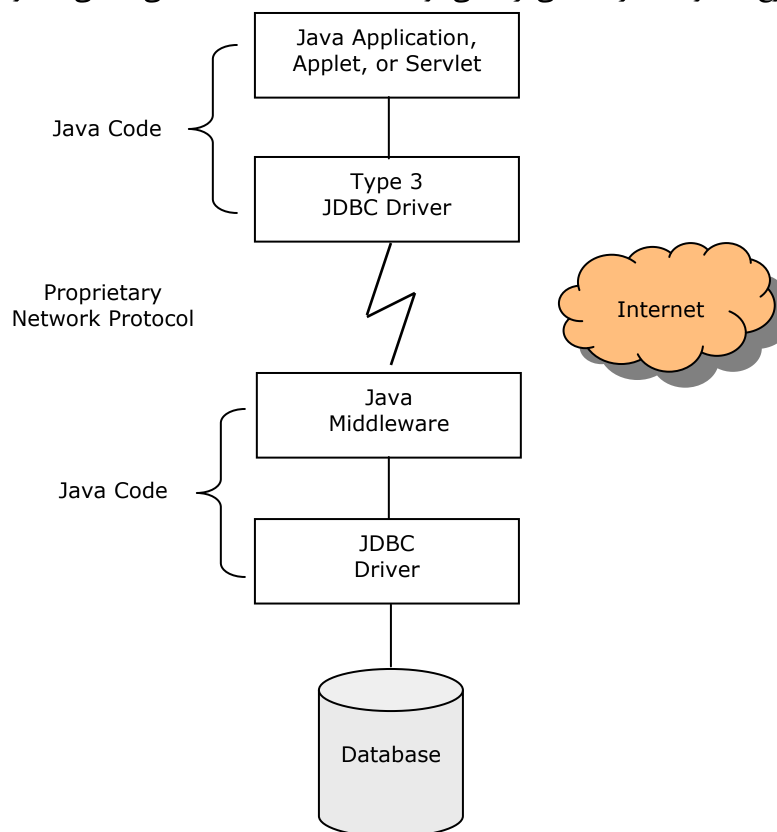
در هر یک از موارد زیر می‌توان از این نوع داریور استفاده کرد.

- به عنوان جایگزینی برای واسط JDBC-ODBC استفاده از درایور نوع دو بهتر از واسط JDBC-ODBC است، چراکه به طور مستقیم با بانک اطلاعاتی ارتباط برقرار می‌کند.
- به عنوان یک راه حل ارزان قیمت البته به شرطی که سیستم بانک اطلاعاتی درایور نوع دو را ارائه دهد. (مانند Oracle، Informix، Sybase و DB2).

3. نوع سه : Java to Proprietary Network Protocol

این نوع درایور JDBC بسیار انعطاف‌پذیر بوده و برای مدل سه لایه و دسترسی از طریق اینترنت قابل استفاده است. درایور نوع سه بطور کامل به زبان جاوا نوشته شده‌است و ارتباط آن با لایه‌های میانی از طریق یک پروتکل انحصاری است که توسط فروشنده بانک اطلاعاتی ارائه می‌شود. این لایه میانی روی سرویس‌دهنده وب یا سرویس‌دهنده بانک اطلاعاتی قرار می‌گیرد.

درایور نوع سه توسط شرکت‌هایی ارائه می‌شود که نسبت به بانک اطلاعاتی خاصی محدود نشده‌اند.



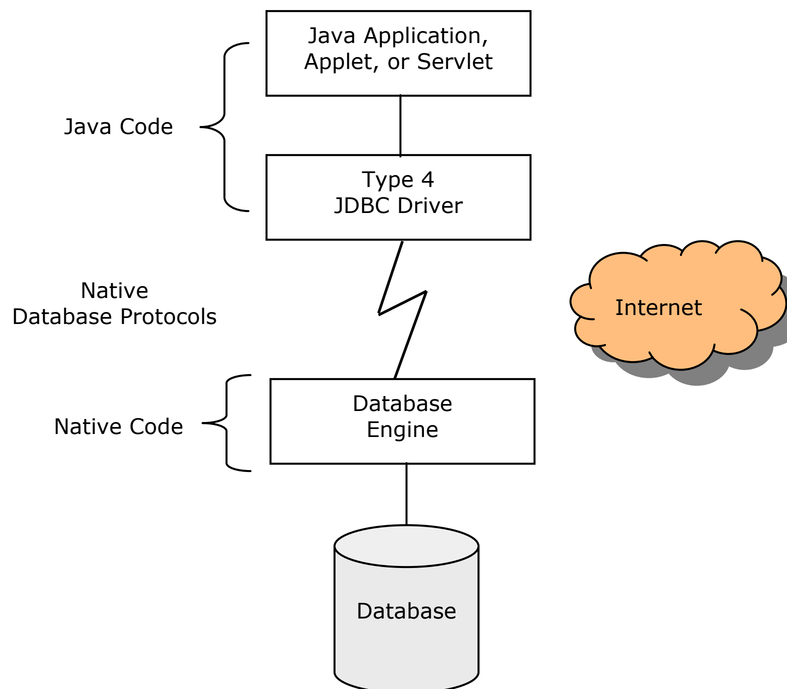
شکل 4-16 درایور نوع سه

در هر یک از موارد زیر می‌توان از این درایور استفاده کرد:

- در اپلت‌ها، چون نیازی به نصب و پیکربندی در دستگاه سرویس‌گیرنده نیست.
- در سیستم‌های امنیتی که بانک اطلاعاتی در لایه میانی حفاظت می‌شود.
- به عنوان یک راه حل انعطاف‌پذیر برای مواردی که از چند نوع بانک اطلاعاتی استفاده می‌شود.

4. نوع چهار: Java to Native Database Protocol

این درایور نیز بطور کامل به زبان جاوا نوشته شده و بطور مستقیم با بانک اطلاعاتی از طریق پروتکل محلی خودش ارتباط برقرار می‌کند و از طریق اینترنت می توان از آن استفاده کرد. کارایی این درایور بیشتر از سایر درایورها بوده و هیچ لایه میانی اعم از کد محلی یا نرم افزار خاصی بین سرویس‌گیرنده و بانک اطلاعاتی وجود ندارد.



شکل 5-16 درایور نوع چهار

در هر یک از موارد زیر می‌توان از این درایور استفاده کرد:
- □ هنگامیکه کارایی بالا بسیار حیاتی باشد.
- □ در محیط‌هایی که فقط از یک نوع بانک اطلاعاتی استفاده می‌شود.
- □ در برنامه‌هایی که از اپلت استفاده می‌شود.

هنگام انتخاب یک درایور باید سرعت، قابلیت حمل و اطمینان آن را درنظر گرفت. برنامه‌های متفاوت نیازهای متفاوتی دارند، به عنوان مثال برنامه‌های مبتنی بر GUI که در سیستم ویندوز NT اجرا می‌شوند، سرعت درایور نوع دوم کارساز است. اپلت برای دور زدن firewall به درایور نوع سوم نیاز دارد و سرولت که در محیط‌های مختلف اجرا می‌شود، نیاز به انعطاف‌پذیری درایور نوع چهارم دارد.
لیستی از درایورهای قابل دسترس JDBC در آدرس زیر موجود است:

http://java.sun.com/products/jdbc/jdbc.dirvers.html

JDBC URL

درایور JDBC از JDBC URL برای شناسایی و برقراری ارتباط با بانک اطلاعاتی استفاده می‌کند. شکل کلی URL به صورت 'jdbc:driver:databasename' است. این استاندارد بسیار ساده است ولی بانک‌های اطلاعاتی از URL هایی با شکل های مختلف برای برقراری ارتباط استفاده می کنند. به عنوان مثال درایور Oracle از URL زیر استفاده می‌کند.

```
jdbc:oracle:thin:@site:port:database
```

یا واسط JDBC-ODBC از URL زیر استفاده می‌کند.

```
jdbc:odbc:datasource[;attribute-name=attribute-value]*
```

ارتباط با بانک اطلاعاتی

شیء java.sql.Connection اطلاعات ارتباط با بانک اطلاعاتی را کپسوله می‌کند و اساس تمام کدهای کنترل JDBC را تشکیل می‌دهد. یک برنامه می‌تواند چندین ارتباط با بانک اطلاعاتی را نگهداری کند، تعداد ارتباط‌های همزمان توسط نوع بانک اطلاعاتی محدود می‌شود. برای مثال نسخه کوچک Oracle تا 50 ارتباط را پشتیبانی می‌کند. بانک‌های اطلاعاتی بزرگ تا چندین هزار ارتباط را پشتیبانی می‌کنند. برای برقراری ارتباط با بانک اطلاعاتی (ایجاد شیء Connection) می‌توان متد()DriverManager.getConnection را فراخوانی کرد.

```
Connection con = DriverManager.getConnection("url", "user", "password");
```

همانطور که ملاحظه می‌شود این متد دارای سه پارامتر JDBC URL، شناسه و رمز عبور کاربر در بانک اطلاعاتی است. در بانک‌های اطلاعاتی که نیاز به ورود صریح (با شناسه کاربر) ندارند، می‌توان شناسه کاربر و رمز عبور را مشخص نکرد. به هنگام فراخوانی متد بالا، DiverManager داریور ذکر شده در url را جستجو کرده و درصورت وجود آن را ثبت و یک شیء Connection برمی‌گرداند. چون متد()getConnection درایورها را به نوبت بررسی می‌کند، باید از بارکردن درایورهای غیرضروری در برنامه خودداری کرد. متد()getConnection دو شکل دیگر نیز دارد که کمتر مورد استفاده قرار می‌گیرد. یک شکل آن فقط پارامتر JDBC URL و شکل دیگر آن از یک پارامتر JDBC URL و یک شیء java.util.Properties استفاده می‌کند که شامل زوج‌های نام و مقدار است. در این فایل حداقل دو زوج username=value و password=value باید مشخص شود.
پس از اتمام عملیات بانکی با فراخوانی متد ()close باید ارتباط با بانک اطلاعاتی را قطع کرد. این کار حافظه اختصاص یافته به شیء Connection را آزاد کرده و مهمتر از آن منابع بانک اطلاعاتی که توسط Connection تخصیص یافته را نیز آزاد می‌کند. این منابع شاید چند بایت بیشتر نباشد ولی همین مقدار کم برای برنامه‌ای مانند سرولت‌ها که نیاز به ایجاد و بستن هزاران Connection دارند، بسیار حیاتی و مهم است. از طرف دیگر چون تعدادی از درایورهای JDBC طوری طراحی شده‌اند که Garbage Collection جاوا نمی‌تواند منابع آن را آزاد سازد، لذا باید از بسته‌شدن Connectionهای غیرضروری اطمینان حاصل کرد.

نسخه JDBC 2.0 که در ادامه به شرح آن پرداخته خواهد شد، یک امکان به نام Connection Pooling ارائه می‌دهد که بوسیله آن می‌توان چندین ارتباط با بانک اطلاعاتی را باز نگه داشت و بین چندین برنامه تقسیم کرد. این امکان برای برنامه‌های بزرگ مانند یک سرولت که روزانه ده هزار تراکنش با بانک اطلاعاتی انجام می‌دهد، بسیار مفید است.

شیء Statement

پس از ایجاد یک Connection و برقراری ارتباط با بانک اطلاعاتی می‌توان دستورات SQL را با استفاده از شیء Statement اجرا کرد. سه نوع مختلف از این شیء در JDBC وجود دارد:

1. Statement: برای اجرای دستورات ساده SQL استفاده می‌شود.
2. PreparedStatement: شامل یک جمله از پیش کامپایل شده‌است که کارایی برنامه را افزایش می‌دهد.
3. CallableStatement: به برنامه‌های JDBC اجازه دسترسی به Stored Procedureهای بانک اطلاعاتی را می‌دهد.

در ادامه فصل به تشریح جملات PreparedStatement و CallableStatement پرداخته می‌شود. برای ایجاد یک شیء Statement باید متد CreateStatement از کلاس Connection را فراخوانی کرد:

```
Statement stmt = con.getConnection ();
```

هر شیء Statement حاوی یک عبارت SQL شامل انتخاب چندین رکورد از یک جدول بانک اطلاعاتی یا عملیاتی دیگری روی بانک اطلاعاتی است. برای بازیابی چندین رکورد از بانک اطلاعاتی، می‌توان از متد executeQuery() شیء Statement استفاده کرد.

```
ResultSet rs = stmt.executeQuery ("SELECT * FROM customers");
```

در مثال بالا برای اجرای دستور SELECT از متد ()executeQuery استفاده شده‌است. این فراخوانی یک شیء از نوع ResultSet را برمی‌گرداند که حاوی رکوردهای حاصل از اجرای دستور SELECT است. Statement دارای متد دیگری به نام ()executeUpdate است که برای اجرای عملیات خاصی مانند INSERT، UPDATE و DELETE روی بانک اطلاعاتی، از آن استفاده می‌شود. متد()executeUpdate یک عدد صحیح برمی‌گرداند که نشان‌دهنده تعداد رکوردهایی است که با اجرای عبارت SQL تغییر کرده‌اند.

اگر عبارت SQL از قبل مشخص نباشد (به عنوان مثال هنگامیکه کاربر عبارت SQL را داخل یک فیلد روی فرم وارد می‌کند) می‌توان از متد()execute استفاده کرد. این متد درصورتیکه نتیجه اجرای عبارت SQL شامل چندین رکورد باشد مقدار true و در غیر اینصورت مقدار false برمی‌گرداند. بنابراین می‌توان از متد()getResultSet برای گرفتن رکورد و از متد()getUpdateCount برای گرفتن تعداد رکوردهای به روز شده، استفاده کرد.

```
Statement unknownSQL = con.createStatement();
if(unknownSQL.execute(sqlString)) {
    ResultSet rs = unknownSQL.getResultSet();
    // display the results
} else {
    System.out.println("Rows updated: " + unknownSQL.getUpdateCount());
}
```

باید به خاطر داشت که شیء Statement یک دستور ساده SQL را اجرا می‌کند. فراخوانی متدهای()executeQuery،()executeUpdate و ()execute به طور ضمنی شیء ResulteSet ایجادشده در جمله قبلی را نامعتبر می‌سازند. اگر برنامه نیاز به اجرای بیش از یک Statement در یک زمان دارد، باید چند شیء Statement ایجاد نمود. به عنوان یک قاعده کلی متد ()close از هر شیء JDBC آن شیء و هر شیء وابسته به آن اعم از Statement ایجاد شده از یک Connection و یا ResultSet ایجاد شده از یک Statement را می‌بندد، اما یک برنامه ساخت یافتهJDBC هر شیء را بطور صریح می‌بندد.

شیءResultSet

هنگام اجرای دستور SQL یک جدول مجازی از رکوردهای منطبق با شرایط جمله پرس و جو به صورت شیء ResultSet برگردانده می‌شوند. برای مثال نتیجه اجرای درخواست زیر:

```
String sql = "SELECT NAME, CUSTOMER_ID, PHONE FROM CUSTOMERS";
```

به صورت زیر خواهد بود.

```
NAME                            CUSTOMER_ID   PHONE
------------------------------- -----------   -------------------
Jane Markham                     1            617 555-1212
Louis Smith                      2            617 555-1213
Woodrow Lang                     3            508 555-7171
Dr. John Smith                   4            (011) 42 323-1239
```

این نوع خروجی برای برنامه‌های جاوا نمی‌تواند خیلی مفید باشد. در مقابل JDBC از رابط java.sql.ResultSet برای کپسوله‌کردن نتایج درخواست پرس و جو به صورت اشیاء و نوع داده جاوا استفاده می‌کند. بفرض ResultSet حاوی جدولی مطابق با نتایج درخواست است. قطعه کد زیر درخواست پرس و جو بالا را کنترل می‌کند.

```
String sql = "SELECT NAME, CUSTOMER_ID, PHONE FROM CUSTOMERS";
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()) {
   System.out.print("Customer #" + rs.getString("CUSTOMER_ID"));
   System.out.print(", " + rs.getString("NAME"));
   System.out.println(", is at " + rs.getString("PHONE");
}
rs.close();
stmt.close();
```

خروجی برنامه به صورت زیر است.

Customer #1, Jane Markham, is at 617 555-1212

```
Customer #2, Louis Smith, is at 617 555-1213
Customer #3, Woodrow Lang, is at 508 555-7171
Customer #4, Dr. John Smith, is at (011) 42 323-1239
```

کد بالا در یک حلقه while برای گرفتن هر رکورد، یک بار متد()next را فراخوانی می‌کند. هنگام ایجاد یک ResultSet موقعیت مکان‌نما، قبل از اولین رکورد است. بدین معنی که برای دسترسی به اولین رکورد باید متد ()next را فراخوانی کرد. به ازای هر بار فراخوانی متد()next، مکان‌نما به رکورد بعدی حرکت می‌کند. اگر هیچ رکوردی برای خواندن وجود نداشته باشد، متد()next مقدار false برمی‌گرداند. در ResultSet ارائه شده در JDBC1.0 مکان‌نما فقط به سمت جلو حرکت می‌کند و هیچ راهی برای بازگشت به رکورد قبلی وجود ندارد، بطوریکه هر رکورد را یک بار می‌توان بازیابی کرد، اما در نسخه JDBC2.0 این محدودیت‌ها برطرف شده‌است.

مقدار هر فیلد یک رکورد را می‌توان با متد()getString بدست آورد. متد()getString یکی از اعضای خانواده متدهای()getXXX است که هر کدام، یک نوع داده‌ای را برمی‌گردانند. دو شکل از متد()getXXX وجود دارد، بطوریکه یکی نام فیلد (مانند "PHONE" و "CUSTOMER-ID") و دیگری شماره اندیس فیلد را به عنوان پارامتر می‌پذیرد. برخلاف اندیس آرایه‌ها در زبان جاوا که از صفر تا n-1 است، اندیس فیلد از یک تا n می‌باشد.

مهمترین متد خانواده()getXXX متد()getObject است که می‌تواند تمام انواع داده را به صورت یک شیء Object برگرداند. برای مثال فراخوانی متد()getObject برای فیلد صحیح یک شیءInteger و برای فیلد تاریخ یک شیء java.sql.Date برمی‌گرداند. برای مواردی که نوع داده برگشتی از متد()getXXX با نوع داده جاوا متفاوت باشد، نوع بازگشتی داخل پرانتز نوشته شده‌است.
نکته : کلاس java.sql.Types یکسری ثابت عددی برای نمایش نوع داده SQL استاندارد تعریف کرده‌است.

جدول 16-1 تطابق نوع داده جاوا و SQL

متد ()getXXX	نوع داده جاوا	نوع داده SQL
getString()	String	CHAR
getString()	String	VARCHAR

LONGVARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean (Boolean)	getBoolean()
TINYINT	Integer (byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
REAL	Float (float)	getFloat()
FLOAT	Double (double)	getDouble()
DOUBLE	Double (double)	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	Java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()
BLOB	Java.sql.Blob	getBlob()
CLOB	Java.sql.Clob	getClob()

این جدول مطابق با مشخصات JDBC بوده و بعضی از درایورها ممکن است مطابق با این نگاشت عمل نکنند. همچنین متد getString() معادل String تمام انواع داده را برمی‌گرداند.

ResultSetهای چندگانه

نوشتن یک عبارت SQL که بیش از یک شیء ResultSet را برگرداند امکان‌پذیر است. شیء Statement دارای متدی به نام getMoreResults() است. فراخوانی این متد بطور ضمنی هر شیء ResultSet موجود را بسته و مجموعه بعدی ResultSet را برمی‌گرداند. اگر ResultSet دیگری قابل دسترسی باشد، متدgetMoreResults() مقدار true و در غیر اینصورت مقدار false برمی‌گرداند. اگر update مدنظر باشد، متد getMoreResults() مقدار false برمی‌گرداند.

مثال قبلی برای کنترل ResultSet چند گانه به شکل زیر تغییر پیدا می‌کند.

```
Statement unknownSQL = con.createStatement();
unknownSQL.execute(sqlString);
while(true){
    rs = unknownSQL.getResultSet();
    if(rs != null)
        // display the results
    else
        // process the update data
    // Advance and quit if done
    if((unknownSQL.getMoreResults() == false) && (unknownSQL.getUpdateCount() == -1))
        break;
}
```

کنترل null

گاهی اوقات فیلدهای بانک اطلاعاتی دارای مقدار null یا خالی می‌باشند. برای کنترل مقدار null و جلوگیری از خطای زمان اجرا، JDBC متد wasNull() را ارائه کرده‌است که بررسی می‌کند که آیا آخرین فیلد خوانده شده مقدار null دارد یا خیر.

```
int numberInStock = rs.getInt("STOCK");
if(rs.wasNull())
    System.out.println("Result was null");
else
    System.out.println("In Stock: " + numberInStock);
```

از متد()getObject نیز می‌توان برای بررسی null بودن یک فیلد استفاده کرد.

```
Object numberInStock = rs.getObject("STOCK");
if(numberInStock == null)
   System.out.println("Result was null");
```

انواع داده‌های حجیم

می‌توان اطلاعات با حجم بالا[1] را نیز از یک ResultSet خواند. این کار می‌تواند به هنگام بازیابی تصاویر و مستندات بزرگ از بانک اطلاعاتی مفید باشد. متدهای ارائه شده ResultSet برای خواندن مقادیر بزرگ عبارتند از ()getAsciiStream، ()getBinaryStream و ()getUnicodeStream که هر کدام نام فیلد یا شماره اندیس را به عنوان پارامتر می‌پذیرند.

در مثال زیر یک تصویر از جدول PICTURES خوانده و روی خروجی (برای مثال ServletOutputStream در یک سرولت جاوا که تصویر را به صورت یک فایل GIF ایجاد می‌کند) نشان می‌دهد.

```
ResultSet rs =  stmt.executeQuery("SELECT IMAGE FROM PICTURES WHERE PID = "
              +  req.getParameter("PID"));

if(rs.next()){
   BufferedInputStream gifData =
               new BufferedInputStream(rs.getBinaryStream("IMAGE"));
   byte[] buf = new byte[4 * 1024]; // 4K buffer
   int len;
   while((len = gifData.read(buf, 0, buf.length)) != -1) {
      out.write(buf, 0, len);
   }
}
```

نسخه JDBC2.0 شیء Blob و Clob را ارائه داده‌است که برای ذخیره و بازیابی انواع داده حجیم مناسب است.

[1] Large Data Types

تاریخ و زمان

JDBC برای ذخیره‌سازی اطلاعات تاریخ و زمان سه کلاس مختلف تعریف کرده‌است :

- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

این سه کلاس متناظر با داده‌های DATE، TIME و TIMESTAMP هستند. کلاس java.util.Date برای سه نوع یادشده مناسب نیست، بنابراین JDBC یک کلاس Date سازگار در بسته java.sql تعریف کرده‌است.

نوع داده DATE در SQL فقط شامل تاریخ بوده، بنابراین کلاس java.sql.Date شامل روز، ماه و سال است. نوع داده java.sql.Time فقط شامل زمان و بدون تاریخ است. نوع داده java.sql.Timestamp شامل تاریخ و زمان با دقت نانو ثانیه است (دقت کلاس Date استاندارد تا میلی ثانیه است).

DBMS های مختلف متدهای متفاوتی برای نمایش اطلاعات تاریخ و زمان دارند. شکل تاریخ، زمان و تاریخ/زمان به صورت زیر است.

```
{d 'yyyy-mm-dd'}
{t 'hh:mm:ss'}
{ts 'yyyy-mm-dd hh:mm:ss.ms.microseconds.ns'}
```

مثال زیر چگونگی استفاده از یک date escape sequence را نشان می‌دهد.

```
dateSQL.execute("INSERT INTO FRIENDS(BIRTHDAY) VALUES ({d '1978-12-14'})");
```

کنترل پیشرفته ResultSet

در JDBC1.0 کارآیی رابط ResultSet بسیار محدود است. این رابط به روزرسانی رکورد، دسترسی تصادفی و برگشت به عقب را پشتیبانی نمی‌کند.

JDBC2.0 یک ResultSet قابل پیمایش و قابل تغییر را برای پیمایش پیشرفته رکورد و تغییر درجای اطلاعات ارائه می‌کند. باوجود قابلیت پیمایش، بجای استفاده از متد next() برای حرکت به رکورد بعدی می‌توان بین رکوردها به جلو و عقب حرکت کرد. در این مبحث سه نوع ResultSet وجود دارد:

- forward-only
- scroll-insensitive
- scroll-sensitive

ResultSet از نوع scroll-insensitive نسبت به تغییر اطلاعات حساس نبوده، در حالیکه نوع scroll-sensitive نسبت به تغییر اطلاعات حساس است. در JDBC2.0 شیء ResultSet قابل به روزرسانی است، از این نظر دو نوع ResultSet وجود دارد:

- **فقط خواندنی** که اجازه تغییر و به روزرسانی اطلاعات را نمی‌دهد.
- **تغییرپذیر** که اجازه تغییر و به روزرسانی اطلاعات را می دهد.

برای ایجاد یک ResultSet از نوع updateable و scroll-sensitive باید دو پارامتر اضافه به متد createStatement() ارسال کرد.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATEABLE);
```

اگر هیچ پارامتری به متدcreateStatement() ارسال نشود، یک ResultSet از نوع فقط خواندنی و رو به جلو تعریف می‌شود. برای پیمایش رکوردهای ResultSet قابل پیمایش باید از

متدهای جدول زیر استفاده نموده. در اینجا نیز همانند JDBC1.0 برای دسترسی به رکوردهای ResultSet ابتدا باید متد()next را فراخوانی کرد.

جدول 16-2 توابع پیمایش رکورد

متد	شرح
first()	رجوع به اولین رکورد
last()	رجوع به آخرین رکورد
next()	رجوع به رکورد بعدی
previous()	رجوع به رکورد قبلی
beforeFirst()	رجوع به قبل از اولین رکورد
afterLast()	رجوع به بعد از آخرین رکورد
absolute(int)	رجوع به شماره رکورد خاص
relative()	حرکت نسبی به عقب یا جلو براساس مثبت یا منفی بودن پارامتر

JDBC2.0 شامل چندین متد است که موقعیت مکان‌نما را اعلام می‌کند. متدهای()isFirst و()isLast اگر مکان‌نما روی اولین یا آخرین رکورد باشد مقدار true برمی‌گردانند. متدهای ()isAfterLast و ()isBeforeFirst به ترتیب اگر مکان‌نما پس از آخرین رکورد و قبل از اولین رکورد باشد مقدار true برمی‌گردانند. با داشتن یک ResultSet قابل تغییر می‌توان اطلاعات رکورد جاری را تغییر داد یا حذف کرد و رکورد جدیدی درج نمود. برای تغییر اطلاعات رکورد جاری از متدهای خانواده()updateXXX استفاده می‌شود. به عنوان مثال برای تغییر فیلد CUSTOMER-ID متعلق به اولین رکورد جدول CUSTOMERS باید به شکل زیر عمل کرد.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
               ResultSet.CONCUR_UPDATEABLE);
ResultSet rs = stmt.executeQuery("SELECT NAME, CUSTOMER_ID FROM CUSTOMERS");
rs.first();
rs.updateInt(2, 35243);
rs.updateRow();
```

در کد بالا ابتدا از متد()first برای حرکت به اولین رکورد و سپس از متد()updateInt برای تغییر مقدار فیلد CUSTOMER-ID استفاده شده‌است. پس از تعیین تغییرات، برای اعمال آن در بانک اطلاعاتی، متد()updateRow فراخوانی می‌شود. اگر قبل از حرکت به سطر بعدی فراخوانی متد()updateRow فراموش شود، تغییرات اعمال نخواهد شد.

اگر نیاز به تغییر چندین فیلد در رکورد است ، با استفاده از چندین متد()updateXXX و یک متد()updateRow این کار قابل انجام است. باید از فراخوانی این متد قبل از حرکت به رکورد دیگر اطمینان حاصل کرد. اگرچه تکنیک درج رکورد جدید مشابه عمل به روزرسانی رکورد است ولی چند تفاوت مهم نسبت به آن دارد. اولین قدم حرکت مکان‌نما به سطر ورودی[1] با فراخوانی متد()moveToInsertRow است. سطر ورودی یک سطر خالی متناظر با فیلدهای رکورد است. پس از حرکت به سطر ورودی از متدهای خانواده()updateXXX برای درج رکورد خالی استفاده می‌شود و در نهایت متد()insertRow رکورد را بطور فیزیکی در بانک اطلاعاتی درج می‌کند.

کد زیرنحوه اضافه کردن یک رکورد به جدول CUSTOMERS را به روش بالا نشان می‌دهد.

```
ResultSet rs = stmt.executeQuery("SELECT NAME, CUSTOMER_ID FROM CUSTOMERS");
rs.moveToInsertRow();
rs.updateString(1, "Tom Flynn");
rs.updateInt(2, 35244);
rs.insertRow();
```

قابل توجه است که در مثال بالا کلیه فیلدهای جدول مقداردهی نشده‌است. فیلدهایی که نمی‌توانند مقدار null بپذیرند، باید مقداردهی شوند. اگر برای فیلدی که نمی‌تواند مقدار null داشته باشد، مقداری مشخص نشود، خطای SQLException رخ خواهد داد. پس از فراخوانی متد()insertRow می‌توان رکورد دیگری درج یا با یکی از متدهای جدول 2-16 روی ResultSet حرکت کرد.

[1] Insert Row

یکی از متدهای پیمایش که در جدول 2-16 ذکرنشده متد()moveToCurrentRow است. این متد مکان‌نما را به رکوردی منتقل می‌کند که قبل از فراخوانی متد()moveToInsertRow بوده‌است. این متد را فقط هنگام درج رکورد جدید می‌توان فراخوانی کرد. حذف رکورد از یک ResultSet قابل‌تغییر بسیار آسان است. کافی است مکان‌نما را به رکوردی منتقل نمود که قرار است، حذف شود و متد()deleteRow را فراخوانی کرد. مثال زیر آخرین رکورد را حذف می‌کند.

```
rs.last();
rs.deleteRow();
```

کنترل خطاها

هر شیء JDBC با خطاهای SQLException مواجه می‌شود. برای مثال خطای ارتباط با بانک، جملات SQL اشتباه و عدم مجوز دسترسی به بانک اطلاعاتی منجر به بروز SQLException می‌شود.

کلاس SQLException از کلاس java.lang.Exception ارث می‌برد و علاوه‌بر این متد جدید به نام ()getNextException تعریف می‌کند. کلاس SQLException دو متد به نام‌های()getSQLState و ()getErrorCode برای گرفتن اطلاعات بیشتر راجع به خطا ارائه می‌دهد. کد برگشتی از متد()getSQLState یکی از کدهای ANSI-92 SQL است. متد()getErrorCode کد مربوط به DBMS بانک اطلاعاتی را برمی‌گرداند، کنترل خطا در بلاک try-catch انجام می‌شود.

```
try {
   // Actual database code
} catch (SQLException e) {
   while(e != null) {
      System.out.println("\nSQL Exception:");
      System.out.println(e.getMessage());
      System.out.println("ANSI-92 SQL State: " + e.getSQLState());
      System.out.println("Vendor Error Code: " + e.getErrorCode());
      e = e.getNextException();
   }
}
```

SQLWarning برای مواردی است که روند اجرای برنامه قطع نمی‌شود و همه چیز درست به نظر می‌رسد. برای مثال ضرب دو عدد از نوع decimal باعث بروز هشدار می‌شود. SQLWarning همان اطلاعات SQLException را نگهداری می‌کند ولی برخلاف شیء SQLException که با بلاک try-catch کنترل می‌شود، این کلاس رابط‌های Connection ، ResultSet، CallableStatement، PreparedStatement و Statement را با متد getWarning() بررسی می‌کند. SQLWarning مشابه SQLException، متدهای getMessage()، getSQLException() و getErrorCode() را پیاده‌سازی کرده‌است. در هنگام دنبال کردن[1] و خطایابی برنامه با متد printWarnings() می‌توان هشدارهای بانک اطلاعاتی را کنترل کرد.

```
void printWarnings(SQLWarning warn) {
   while (warn != null) {
      System.out.println("\nSQL Warning:");
      System.out.println(warn.getMessage());
      System.out.println("ANSI-92 SQL State: " + warn.getSQLState());
      System.out.println("Vendor Error Code: " + warn.getErrorCode());
      warn = warn.getNextWarning();
   }
}
```

سپس می‌توان همانند کد زیر از متد printWarnings() استفاده کرد.

```
printWarnings(stmt.getWarnings());
printWarnings(rs.getWarnings());
// Rest of database code
```

شیء PreparedStatement

این شیء بسیار مشابه شیء Statement می‌باشد و هر دو دستورات SQL را اجرا می‌کنند. شیء PreparedStatement این امکان را فراهم می‌کند که دستورات SQL را با تعیین

[1] Debugging

پارامترهای مورد نظر از قبل کامپایل و آن را چندین بار اجرا کرد. از آنجاییکه بررسی و پردازش جملات SQL زمان قابل توجهی را صرف خود می‌کند، کامپایل کردن جملات SQL خارج از بانک اطلاعات یکی از راه‌های بالا بردن کارآیی بانک اطلاعاتی محسوب می‌شود. همانند شیء Statement یک شیء PreparedStatement نیز از شیء Connection ایجاد می‌شود. در این حالت جمله SQL در زمان ایجاد و نه در زمان اجرا توسط متد()preparedStatement شیء Connection مشخص می‌شود.

```
PreparedStatement pstmt = con.prepareStatement(
"INSERT INTO EMPLOYEES (NAME, PHONE) VALUES (?, ?)");
```

این دستور SQL یک رکورد جدید به جدول EMPLOYEE با مشخص کردن مقدار فیلدهای NAME و PHONE اضافه می‌کند. از آنجاییکه PreparedStatement چندین بار قابل اجرا است، مقدار فیلدها در داخل بدنه SQL مشخص نشده بلکه از کاراکتر '?' برای تعیین پارامتر استفاده شده‌است. البته قبل از اجرای SQL باید مقدار پارامتر را مشخص نمود.

```
pstmt.clearParameters();
pstmt.setString(1, "Jimmy Adelphi");
pstmt.setString(2, "201 555-7823");
pstmt.executeUpdate();
```

در مثال بالا ابتدا، پارامترهای از قبل تعیین‌شده با متد()clearParameters پاک شده و سپس مقدار دو پارامتر با متد ()setString مشخص شده‌است. رابط PreparedStatement چندین متد()setXXX برای تعیین پارامترها تعریف کرده‌است، برا ی مشاهده لیست کامل این متدها باید به مرجع java.sql مراجعه شود. متد ()setObject می‌تواند هر نوع شیء جاوا را در بانک اطلاعاتی درج کند و به شکل‌های زیر می‌توان از آن استفاده کرد.

```
setObject(int parameterIndex, Object x, int targetSqlType, int scale)
setObject(int parameterIndex, Object x, int targetSqlType)
setObject(int parameterIndex, Object x)
```

از PreparedStatement می‌توان برای درج مقدار null در بانک اطلاعاتی با فراخوانی متد ()setNull یا تعیین مقدار null در متد()setXXX استفاده کرد. همانطور که ملاحظه می‌شود در

مثال زیر پارامتر اول یک شیء Integer با مقدار یک و پارامتر دوم یک شیء Varchar با مقدار null می‌باشد.

```
Integer i = new Integer(32);
pstmt.setObject(1, i, Types.INTEGER);
pstmt.setObject(2, null, Types.VARCHAR);
// or pstmt.setNull(2, Types.VARCHAR);
```

به روز رسانی گروهی[1]

استاندارد اولیه JDBC برای بارکردن حجم زیاد اطلاعات به بانک اطلاعاتی حتی درصورت استفاده از شیء PreparedStatement خیلی کارآمد نیست. اگر برنامه بخواهد 10000 رکورد به بانک اطلاعاتی اضافه کند، سرعت و کارآیی بانک اطلاعاتی به شدت تنزل پیدا می‌کند. متد جدید Statement به نام()addBatch امکان اجرای چندین دستور SQL را به صورت Batch فراهم می‌کند. پس از ایجاد Statement و قبل از اجرای آن باید()addBatch را فراخوانی کرد.

```
con.setAutoCommit(false); // If some fail, we want to rollback the rest
Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO CUSTOMERS VALUES (1, 'J Smith', '617 555-1323'");
stmt.addBatch("INSERT INTO CUSTOMERS VALUES (2, 'A Smith', '617 555-1132'");
stmt.addBatch("INSERT INTO CUSTOMERS VALUES (3, 'C Smith', '617 555-1238'");
stmt.addBatch("INSERT INTO CUSTOMERS VALUES (4, 'K Smith', '617 555-7823'");
int[] upCounts = stmt.executeBatch();
con.commit();
```

قابل توجه است که قطعیت اتوماتیک کار[2] غیرفعال شده‌است. این کار برای بی اثر کردن[3] اجرای دستورات SQL که با مشکل مواجه شده‌اند ضروری است.

[1] Batch Update
[2] auto-commit
[3] rollback

پس از فراخوانی چندین متد()addBatch، متد()executeBatch برای اجرای دستورات SQL فراخوانی می‌شود. این نوع دستورات به ترتیب اضافه شدن به Batch اجرا می‌شوند. متد()executeBatch یک آرایه از تعداد رکوردهایی که با اجرای هر دستور SQL به روزرسانی شده‌است را برمی‌گرداند. برای حذف یک دستور از لیست دستورات در انتظار، باید از متد()clearBatch قبل از فراخوانی متد()executeUpdate استفاده کرد.

نکته : فقط دستورات CREATE، DROP، INSERT، UPDATE و DELETE را می‌توان به صورت Batch اجرا کرد. اگر دستور SELECT در Batch قرار گیرد، خطای SQLException رخ می‌دهد. اگر یکی از دستورات Batch به هر دلیلی اجرا نشود، متد()executeBatch با خطای BatchUpdateException مواجه خواهد شد.

نحوه استفاده از متد()addBatch در جملات PreparedStatement و CallableStatement کمی متفاوت است. برای استفاده از این متد در PreparedStatement یک بار دستور SQL تعریف می‌شود و به ازای هر رکورد پارامترهای آن مشخص می‌شود و در نهایت متد()executeBatch بدون پارامتر فراخوانی می‌شود.

```
con.setAutoCommit(false); // If some fail, we want to rollback the rest
PreparedStatement stmt =
        con.prepareStatement("INSERT INTO CUSTOMERS VALUES (?,?,?)");
stmt.setInt(1,1);
stmt.setString(2, "J Smith");
stmt.setString(3, "617 555-1323");
stmt.addBatch();

stmt.setInt(1,2);
stmt.setString(2, "A Smith");
stmt.setString(3, "617 555-1132");
stmt.addBatch();

int[] upCounts = stmt.executeBatch();
con.commit();
```

از این مکانیزم می‌توان برای CallableStatement که مخصوص فراخوانی Stored Procedure است نیز استفاده کرد به شرطی که هر Stored Procedure یک مقدار (تعداد رکوردهای به روز شده) را برگرداند و هیچ پارامتری از نوع OUT یا INOUT نداشته باشد.

BLOB و CLOB

افزایش حجم اطلاعات ذخیره شده در بانک‌های اطلاعاتی، عرضه کنندگان بانک اطلاعاتی را به سمت پشتیبانی از LOB[1] سوق داد. دو نوع LOB به نام‌های BLOB[2] و CLOB[3] برای ذخیره‌سازی اطلاعات باینری یا کاراکتری با حجم بالا وجود دارد. پشتیبانی از LOB بستگی به نوع بانک اطلاعاتی دارد. بعضی از بانک‌های اطلاعاتی LOB را پشتیبانی نمی‌کنند و برخی نیز دارای نوع دیگر آن (BINARY و LONG RAW) هستند.

در JDBC1.0 با متدهای زیر می‌توان به فیلدهای CLOB وBLOB دسترسی پیدا کرد.

- getBinaryStream()
- getAsciiStream()
- getCharacterStream()

در JDBC2.0 رابط ResultSet شامل دو متد()getBlob و ()getClob است که به ترتیب شیء BLOB و CLOB را برمی‌گردانند.

دسترسی به محتوای شیء BLOB از طریق متد()getBinaryStream و شیء CLOB از طریق متد ()getCharacterStream امکان‌پذیر است. برای دسترسی مستقیم می‌توان از متد()getBytes برای شیء BLOB و متد()getCharacterStream برای شیء Clob نیز استفاده کرد. اطلاعات یک فیلد CLOB براحتی با شیء CLOB و متد()getCharacterStream قابل بازیابی است.

```
String s;
Clob clob = blobResultSet.getBlob("CLOBFIELD");
BufferedReader clobData = new BufferedReader(clob.getCharacterStream());
while((s = clobData.readLine()) != null)
   System.out.println(s);
```

[2] Binary Large Object
[3] Character Large Object

علاوه بر این می‌توان فیلدهای BLOB و CLOB را هنگام استفاده از شیء PrepareStatement با متدهای ()setBlob و ()setClob مقداردهی کرد با توجه به اینکه متدهای()updateBlob و()updateClob برای تعیین مقادیر شیء BLOB و CLOB درنظر گرفته شده‌است.

شبه داده

بیشتر برنامه‌های JDBC برای کار با یک بانک اطلاعاتی مشخص شده‌است، یعنی برنامه دقیقاً می‌داند با چه نوع داده اطلاعاتی درگیر است. بعضی از برنامه‌ها نیاز به دریافت دینامیک اطلاعات درباره ساختار ResultSet و پیکربندی بانک اطلاعاتی دارند که به این اطلاعات شبه داده[1] گفته می‌شود. JDBC دو کلاس DataBaseMetaData و ResultSetMetaData را برای این نوع داده درنظر گرفته‌است. اگر قرار است برنامه‌ایی تولید شود که در چندین بستر و بانک اطلاعاتی مختلف کار کند باید از این کلاس‌ها استفاده کند.

DatabaseMetaData

اطلاعات عمومی راجع به ساختار بانک اطلاعاتی را می‌توان با استفاده از رابط java.sql.DatabaseMetaData بازیابی کرد با بهره‌گیری کامل از این کلاس می‌توان دستورات SQL را با توجه به پشتیبانی درایور JDBC و نوع بانک اطلاعاتی سازگار ساخت. شیء DatabaseMetaData را می‌توان با متد()getMetaData از Connection ایجاد کرد.

```
DatabaseMetaData dbmeta = con.getMetaData();
```

[1] Metadata

```
rs.last();
rs.deleteRow();
```

کلاس DatabaseMetaData چندین متد برای گرفتن اطلاعات واقعی درباره پیکربندی بانک اطلاعاتی ارائه می‌دهد. بعضی از این متدها رشته(getURL())، برخی دیگر مقدار منطقی(nullAreSortedHigh()) و برخی نیز مقدار صحیح(getMaxConnection()) برمی‌گردانند. متدهای getTableTypes()، getColumnTypes() و getPrivileges() نیز شئیی از نوع ResultSet برمی‌گردانند. برای مثال، متد(getTableTypes() لیست تمام جدول های بانک اطلاعاتی را به صورت ResultSet برمی‌گرداند.

تعدادی از متدهای کلاس DatabaseMetaData یک عبارت را به عنوان آرگومان برای جستجو در اطلاعات MetaData می پذیرند. علامت '%' برای هر تعداد کاراکتر و علامت '_' برای تطبیق یک کاراکتر است. بنابراین عبارت %CUSTOMER% با مقادیر NEW-CUSTOMER و CUSTOMER و عبارت CUSTOMER% با مقادیر CUSTOMER و CUSTOMERS تطابق دارد.

مثال زیر یک برنامه ساده برای نمایش اطلاعات پایه شامل لیست جدول های و اندیس‌ها است. فرض‌بر این است که JDBC تمام دستورات DatabaseMetaData را پشتیبانی می‌کند.

```
import java.sql.*;
import java.util.StringTokenizer;

public class DBViewer {
   final static String jdbcURL = "jdbc:odbc:customerdsn";
   final static String jdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver";

   public static void main(java.lang.String[] args) {
      System.out.println("--- Database Viewer ---");
      try {
         Class.forName(jdbcDriver);
         Connection con = DriverManager.getConnection(jdbcURL, "", "");
         DatabaseMetaData dbmd = con.getMetaData(  );
         System.out.println("Driver Name: " + dbmd.getDriverName(  ));
         System.out.println("Database Product: " + dbmd.getDatabaseProductName());
         System.out.println("SQL Keywords Supported:");
         StringTokenizer st = new StringTokenizer(dbmd.getSQLKeywords(), ",");
         while(st.hasMoreTokens())
            System.out.println(" " + st.nextToken());

         // Get a ResultSet that contains all of the tables in this database
         // We specify a table_type of "TABLE" to prevent seeing system tables,
         // views and so forth
         String[] tableTypes = {"TABLE"};
         ResultSet allTables = dbmd.getTables(null, null, null, tableTypes);
         while(allTables.next()){
            String table_name = allTables.getString("TABLE_NAME");
            System.out.println("Table Name:" + table_name);
            System.out.println("Table Type:" + allTables.getString("TABLE_TYPE"));
            System.out.println("Indexes: ");
            // Get a list of all the indexes for this table
            ResultSet indexList = dbmd.getIndexInfo(null,null,table_name,false,false);
            while(indexList.next()) {
               System.out.println(" Index Name: " + indexList.getString("INDEX_NAME"));
               System.out.println(" Column Name: " + indexList.getString("COLUMN_NAME"));
            }
            indexList.close();
         }
         allTables.close();
         con.close();
      } catch (ClassNotFoundException e) {
         System.out.println("Unable to load database driver class");
      } catch (SQLException e) {
         System.out.println("SQL Exception: " + e.getMessage());
```

کد 16-2 نمایش اطلاعات پایه

اگر مثال بالا برای بانک اطلاعاتی Microsoft Access و با استفاده از واسط JDBC-ODBC اجرا شود، خروجی به صورت زیر خواهد بود.

```
--- Database Viewer ---
Driver Name: JDBC-ODBC Bridge (odbcjt32.dll)
Database Product: ACCESS
SQL Keywords Supported:
 ALPHANUMERIC
 AUTOINCREMENT
 BINARY
 BYTE
 FLOAT8
 ...
Table Name: Customers
Table Type: TABLE
Indexes:
 Index Name: PrimaryKey
 Column Name:CustNo
 Index Name: AddressIndex
 Column Name:Address
 ...
```

ResultSetMetaData

رابط ResultSetMetaData اطلاعاتی راجع به ساختار یک ResultSet ارائه می‌دهد. این اطلاعات شامل تعداد فیلدهای قابل دسترس، نام فیلد و نوع داده فیلد است. مثال زیر یک برنامه کوچک برای نمایش محتویات یک جدول و نوع داده هر ستون است.

```
import java.sql.*;

public class TableViewer {

    final static String jdbcURL = "jdbc:oracle:customerdb";
    final static String jdbcDriver = "oracle.jdbc.driver.OracleDriver";
    final static String table = "CUSTOMERS";

    public static void main(java.lang.String[] args) {
        System.out.println("--- Table Viewer ---");
        try {
            Class.forName(jdbcDriver);
            Connection con = DriverManager.getConnection(jdbcURL, "", "");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM "+ table);
            ResultSetMetaData rsmd = rs.getMetaData();
            int columnCount = rsmd.getColumnCount();
```

```
            for(int col = 1; col <= columnCount; col++) {
                System.out.print(rsmd.getColumnLabel(col));
                System.out.print(" (" + rsmd.getColumnTypeName(col)+")");
                if(col < columnCount)
                    System.out.print(", ");
            }
            System.out.println();
            while(rs.next()) {
                for(int col = 1; col <= columnCount; col++) {
                    System.out.print(rs.getString(col));
                    if(col < columnCount)
                        System.out.print(", ");
                }
                System.out.println();
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (ClassNotFoundException e) {
            System.out.println("Unable to load database driver class");
        } catch (SQLException e) {
            System.out.println("SQL Exception: " + e.getMessage());
        }
    }
}
```

کد 16-3 نمایش اطلاعات جدول

متدهای کلیدی رابط ResultSet عبارتند از :

- getColumnTypeName()
- getColumnLabel()
- getColumnCount()

قابل توجه است که نوع داده‌ایی که با متد()getColumnTypeName برگردانده می‌شود وابسته به نوع بانک اطلاعاتی است (برای مثال بانک اطلاعاتی Oracle مقدار VARCHAR و بانک

اطلاعاتی Microsoft Access مقدار TEXT را برای یک فیلد از نوع String برمی‌گرداند) . خروجی مثال بالا چنین است.

```
--- Table Viewer ---
CustNo (SHORT), CustName (VARCHAR), CustAddress (VARCHAR)
1, Jane Markham, 12 Stevens St
2, Louis Smith, 45 Morrison Lane
3, Woodrow Lang, 4 Times Square
```

تراکنش

یک تراکنش[1] گروهی از دستورالعمل‌های مستقل است که باید با هم اجرا شوند. با درنظر گرفتن پشتیبانی بانک اطلاعاتی، تراکنش‌ها اجازه می‌دهند یک یا چند عملیات بانک اطلاعاتی را به یک واحد عملیاتی تبدیل و به یکباره اجرا نمود. اگر برنامه نیاز دارد که چندین دستور SQL بطور کامل و با هم اجرا شوند باید از تراکنش‌ها استفاده کند. به عنوان مثال می‌توان در سیستم مدیریت اموال، انتقال اقلام کالا از جدول INVENTORY به جدول SHIPPING را نام برد.

کار با تراکنش‌ها چند مرحله دارد : شروع تراکنش، انجام عملیات تراکنش، commit کردن تراکنش درصورتیکه تمام دستورات با موفقیت انجام شود و یا roll back کردن تراکنش درصورتیکه اجرای یک یا چند دستور دچار مشکل شود. قابلیت roll back کردن تراکنش‌ها یک ویژگی کلیدی است، بدین معنی که اگر اجرای یکی از دستورات SQL دچار خطا شود، کل عملیات roll back شده و هیچ تاثیری بر بانک اطلاعات نمی‌گذارد. به عنوان مثال جدول INVENTORY بدهکار می‌شود، اما جدول SHIPPING بستانکار نمی‌شود.

تراکنش‌ها می‌توانند در سطوح مختلف ایزوله عمل کنند. در یک سطح ایزوله درصورتیکه تراکنش با موفقیت commit شود، نتایج تمام دستورات SQL برای سیستم قابل رویت است. به عبارت دیگر هرگز کاهش لیست موجودی قبل از به روزرسانی اطلاعات فروش مشاهده نخواهد شد.

شیء Connection در JDBC مسئول مدیریت تراکنش‌ها است. در JDBC بطور پیش‌فرض Connection جدید تراکنش‌ها را به صورت auto-commit تعریف می‌کند. یعنی اینکه هر دستور

[1] Transaction

SQL در یک تراکنش مستقل اجرا و بلافاصله commit می‌شود. برای انجام یک تراکنش که از چندین دستور SQL تشکیل شده‌است باید متد()setAutoCommit را با پارامتر false فراخوانی کرد، پس از اجرای دستورات SQL باید تراکنش را با فراخوانی متد()commit انجام داد یا تراکنش را با فراخوانی متد()rollback بی تاثیر کرد.

```
try {
   con.setAutoCommit(false);
   // run some SQL
   stmt.executeUpdate("UPDATE INVENTORY SET ONHAND = 10 WHERE ID = 5");
   stmt.executeUpdate("INSERT INTO SHIPPING (QTY) VALUES (5)");
   con.commit();
} catch (SQLException e) {
   con.rollback(); //undo the results of the transaction
}
```

هنگامیکه متد()setAutoCommit با پارامتر false فراخوانی شود، متدهای ()commit یا()rollback باید در انتهای تراکنش ذکر شوند، درغیر اینصورت تغییرات اعمال شده بی اثر خواهند شد.

JDBC پنج سطح ایزوله تراکنش را برای کنترل ناسازگاری بین بانک اطلاعاتی و تراکنش‌ها پشتیبانی می‌کند، این سطوح توسط همه بانک‌های اطلاعاتی پشتیبانی نمی‌شود. سطح پیش‌فرض ایزوله بستگی به نوع درایور و بانک اطلاعاتی دارد. هر چه سطح ایزوله بالاتر باشد کارآیی بانک کاهش می‌یابد. این پنج سطح به صورت ثابت در رابط Connection تعریف شده‌است:

- **TRANSACTION-NAME** تراکنش غیر فعال بوده یا پشتیبانی نمی‌شود.

- **TRANSACTION-READ-UNCOMMITLED** در این سطح سایر تراکنش‌ها می‌توانند نتایج دستورات SQL تراکنش اصلی را قبل از commit شدن آن مشاهده نمایند. اگر تراکنش اصلی rollback شود اطلاعات سایر تراکنش‌ها نادرست (نامعتبر) خواهد شد.

- **TRANSACTION-READ-COMMITLED** تراکنش اصلی اجازه خواندن اطلاعات رکوردها را قبل از commit شدن به سایر تراکنش‌ها نمی‌دهد.

- **TRANSACTION-REPEATABIE-READ** از خواندن تکراری داده‌ها محافظت می‌کند. اگر یک تراکنش، رکوردی را بخواند که توسط تراکنش دیگر تغییر کرده و دوباره به رکورد مورد نظر رجوع کند، تغییرات را مشاهده نخواهد کرد.

- **TRANSACTION-SERINLIZABLE** تمام ویژگی‌های سطح TRANSACTION-REPEATABIE-READ را دارا می‌باشد و از درج رکورد جدید جلوگیری می‌کند. بدین معنی که اگر تراکنش به یک مجموعه از رکوردها دسترسی داشته باشد و تراکنش دیگری رکورد جدیدی به این مجموعه اضافه کند و تراکنش اول دوباره به این مجموعه رکورد رجوع کند، رکورد جدید را نمی‌بیند.

سطح ایزوله یک تراکنش با متد()setTransactionIsoloation مشخص می‌شود. مثال:

```
con.setTransactionIsoloation(TRANSACTION-READ-COMMITLED);
```

می‌توان از متدهای کلاس DatabaseMetaData برای تعیین سطح ایزوله حمایت شده توسط بانک اطلاعاتی استفاده کرد. این متدها عبارتند از:

- getDefaultTransactionIsolation()
- supportsTransactions()
- supportsTransactionIsolationLevel()
- supportsDataDefinitionAndDataManipulationTransaction()

Stored Procedure

بیشتر بانک‌های اطلاعاتی دارای یک زبان برنامه‌نویسی داخلی (برای مثال PL/SQL) هستند. این زبان به برنامه‌نویس اجازه می‌دهد یکسری از دستورات SQL را بطور مستقیم در بانک اطلاعاتی ذخیره کرده و در برنامه کاربردی آنها را فراخوانی نماید. مزیت این روش آن است که دستورات SQL یکبار در بانک اطلاعاتی ذخیره و کامپایل شده و توسط برنامه‌های مختلف در بسترهای متفاوت فراخوانی می‌شوند.

همچنین روش مذکور کد برنامه‌ها را از پرداختن به ساختار جداول بانک اطلاعاتی بی نیاز می‌سازد. تمام دستورات SQL در داخل Stored Procedure ذخیره می‌شوند و تنها درصورتی باید اصلاح شوند که ساختار جداول مورد استفاده درStored Procedure تغییر کرده باشد. در مثال زیر یک Stored Procedure در بانک اطلاعاتی Oracle مشاهده می‌شود.

```
CREATE OR REPLACE PROCEDURE sp_interest
(id IN INTEGER
balance IN OUT FLOAT) IS
BEGIN
  SELECT balance  INTO balance FROM accounts
   WHERE account_id = id;

  balance:= balance + balance * 0.03;

  UPDATE accounts SET
     balance = balance
  WHERE account_id = id;

END;
```

این Stored Procedure دو پارامتر ورودی به نام id و balance می‌گیرد و مقدار فیلد balance را به روز تغییر داده و برمی‌گرداند. رابط CallableStatement برای فراخوانی Stored Procedure در JDBC طراحی شده‌است، کلاس Connection متد()prepareCall را که بسیار مشابه متد()prepareStatement است، برای ایجاد یک prepareStatement ارائه می‌کند. بدلیل اینکه هر بانک اطلاعاتی قواعد نگارش مخصوص بخود را برای دسترسی به Stored Procedure دارد، JDBC یکسری قواعد استاندارد برای این کار با استفاده از CallableStatement ارائه می‌دهد. نحوه نگارش Stored Procedure فاقد ResultSet، بدین صورت است.

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

نحوه نگارش Stored Procedure دارای ResultSet نیز بدین صورت است.

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

این نحوه نگارش کاراکتر '?' محل پارامتر یا مقدار بازگشتی Stored Procedure را مشخص می‌کند. JDBC مسئول تبدیل قواعد نگارش استاندارد به قواعد مختص همه بانک اطلاعاتی است.

در قطعه کد زیر استفاده از CallableStatement برای فراخوانی sp-interest نشان داده شده‌است.

```
CallableStatment cstmt = con.prepareCall("{call sp_interest(?,?)}");
cstmt.registerOutParameter(2, Types.FLOAT);
cstmt.setInt(1, accountID);
cstmt.setFloat(2, 2343.23);
cstmt.execute();
out.println("New Balance:" + cstmt.getFloat(2));
```

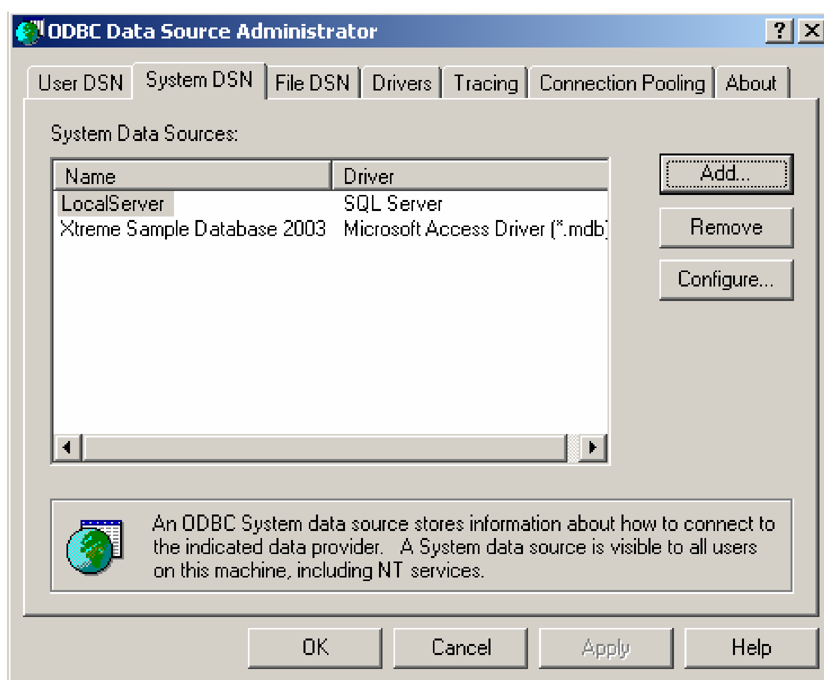
پارامتر خروجی است. از متد ()registerOutParameter برای تعریف پارامتر خروجی استفاده می‌شود. پارامترهای ورودی با متدهای خانواده ()setXXX تعریف می‌شوند. برای گرفتن مقدار پارامتر خروجی از متدهای()getXXX استفاده می‌شود.

مثال JDBC

برای آشنایی با تمام مراحل اولیه استفاده از JDBC به تشریح یک مثال ساده پرداخته می‌شود که به بانک اطلاعاتی Microsoft Access از طریق واسط JDBC-ODBC وصل شده، یک درخواست پرس و جو را برای جدول Employee اجرا و نتایج آن را نمایش داده و درنهایت ارتباط را قطع می‌کند. چون از واسط JDBC-ODBC برای ارتباط با بانک اطلاعاتی Microsoft Access استفاده می‌شود (باید Microsoft Office روی کامپیوتر مورد نظر نصب شده باشد)، ابتدا باید یک DSN[1] تعریف کرد. برای سیستم‌عامل ویندوز مراحل تعریف DSN شرح داده خواهد شد، برای سیستم‌عامل UNIX فایل پیکربندی odbc.init را به صورت دستی باید تغییر داد.

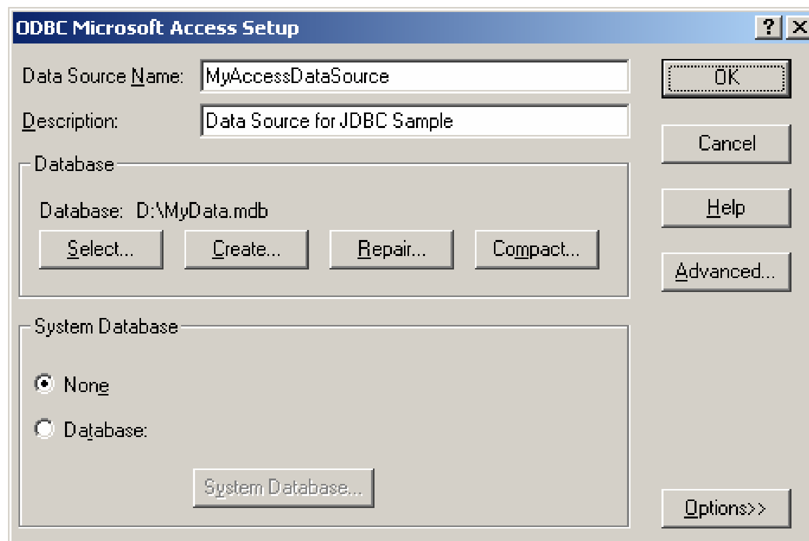
[1] Data Source Name

برای اجرای برنامه ODBC در ویندوز از منوی Start گزینه setting و سپس Control Panel انتخاب می‌شود. در پنجره Control Panel گزینه ODBC یا ODBC32 را باید انتخاب کرد. شکل زیر پنجره ODBC را نمایش می‌دهد.



شکل 6-16 پنجره ODBC ویندوز

برای تعریف DSN جدید روی دکمه‌Add کلیک کرده (حتماً قسمت System DSN انتخاب گردد) سپس در لیست System Data Source لیستی از درایورهای ODBC نصب شده روی کامپیوتر مشاهده می‌شود (این لیست از فایل odbcinst.init استفاده می‌کند). پس از انتخاب گزینه Microsoft Access پنجره مشابه شکل زیر نمایش داده می‌شود.



شکل 7-16 پنجره تعریفODBC برای بانک اطلاعاتی Access

حال باید نام DSN و سایر اطلاعات مورد نیاز را برای بانک اطلاعاتی موردنظر وارد کرد. در این مثال، عبارت "MyAccessDataSource" به عنوان نام DSN بانک اطلاعاتی MyData.mdb که حاوی جدول Employee است، وارد شده‌است. اگر از واسط JDBC-ODBC در یک سرولت برای اجرا در بعضی از نسخه‌های ویندوز (مانند NT) استفاده می‌شود، ODBC از User DSN و System DSN استفاده می‌کند. یک System DSN توسط هر برنامه‌ای که به صورت NT Service نصب شده باشد، قابل استفاده است. درحالیکه یک User DSN فقط برای برنامه‌های کاربر جاری سیستم قابل استفاده است. بعضی از حامل‌های سرولت به عنوان یک NT Service نصب می‌شوند و فقط به اطلاعات System DSN دسترسی دارند.

کد زیر برنامه SimpleQuery.java بوده که اطلاعات جدول Empolyee را از بانک اطلاعاتی Access نمایش می‌دهد.

```
package com.omh.db;
import java.sql.*;

/**
* This simple application will connect to a Microsoft Access
* database using the JDBC-ODBC Bridge, execute a query against
* an employee database, display the results, and then perform
* all of the necessary cleanup
*/

public class SimpleQuery {
   /**
   * Main entry point for the application
   */
   public static void main(String args[]){
      try {
         // Perform the simple query and display the results
         performQuery();
      } catch (Exception ex){
          ex.printStackTrace();
      }
   }

   public static void performQuery() throws Exception {

      // The name of the JDBC driver to use
      String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";

      // The JDBC connection URL
      String connectionURL = "jdbc:odbc:MyAccessDataSource";

      // The JDBC Connection object
      Connection con = null;

      // The JDBC Statement object
      Statement stmt = null;

      // The SQL statement to execute
      String sqlStatement = "SELECT Empno, Name, Position FROM Employee";
      // The JDBC ResultSet object
      ResultSet rs = null;
      try {
         System.out.println("Registering " + driverName);
         // Create an instance of the JDBC driver so that it has
         // a chance to register itself
         Class.forName(driverName).newInstance();
         System.out.println("Connecting to " + connectionURL);
         // Create a new database connection. We're assuming that
         // additional properties (such as username and password)
         // are not necessary
         con = DriverManager.getConnection(connectionURL);
         // Create a statement object that we can execute queries
         // with
         stmt = con.createStatement();
         // Execute the query
         rs = stmt.executeQuery(sqlStatement);
         // Process the results. First dump out the column
         // headers as found in the ResultSetMetaData
         ResultSetMetaData rsmd = rs.getMetaData();
         int columnCount = rsmd.getColumnCount();
```

```
            System.out.println("");
            String line = "";
            for(int i=0; i < columnCount; i++){
                if (i > 0){
                    line += ", ";
                }
                // Note that the column index is 1-based
                line += rsmd.getColumnLabel(i + 1);
            }
            System.out.println(line);
            // Count the number of rows
            int rowCount = 0;
            // Now walk through the entire ResultSet and get each
            // row
            while (rs.next()){
                rowCount++;
                // Dump out the values of each row
                line = "";
                for (int i = 0; i < columnCount; i++){
                    if (i > 0) {
                        line += ", ";
                    }
                    // Note that the column index is 1-based
                    line += rsmd.getColumnLabel(i + 1);
                }
                System.out.println(line);
                // Count the number of rows
                rowCount = 0;
                // Now walk through the entire ResultSet and get each
                // row
                while (rs.next()) {
                    rowCount++;
                    // Dump out the values of each row
                    line = "";
                    for (int i = 0; i < columnCount; i++){
                        if (i > 0){
                            line += ", ";
                        }
                        // Note that the column index is 1-based
                        line += rs.getString(i + 1);
                    }
                    System.out.println(line);
                }
                System.out.println("" + rowCount + " rows, " +
                columnCount + " columns");
            }
        } finally {
            // Always clean up properly!
            if (rs != null){
                rs.close();
            }
            if (stmt != null){
                stmt.close();
            }
            if (con != null){
                con.close();
            }
        }
    }
}
```

کد 4-16 نمایش اطلاعات

خروجی:

```
Registering sun.jdbc.odbc.JdbcOdbcDriver
Connecting to jdbc:odbc:MyAccessDataSource
Empno, Name, Position
1, Nebby K. Nezzer, President
2, Mr. Lunt, Foreman
3, Rack, Jr. Executive
4, Shack, Jr. Executive
5, Benny, Jr. Executive
6, George, Security Guard
7, Laura, Delivery Driver
7 rows, 3 columns
```

خلاصه

- اکثر جملات SQL می‌توانند با کلیه درایورها پردازش شوند، به هرحال برخی از جملات (نظیر جملات متکی بر cursor) ممکن است با کلیه درایورها یا بانک‌های اطلاعاتی کار نکنند.
- با استفاده از API جاوا می‌توان درخواست‌های ساده و پیچیده و کلیه انواع به روزرسانی بانک اطلاعاتی را انجام داد.
- درایورهای JDBC متدها و کلاس‌هایی تهیه کرده‌است که می‌تواند جملات را برای استفاده‌های بعدی کامپایل کند (کلاس PreparedStatement)، جملات ناشناخته SQL را اجرا کند (متد execute())، چندین عملیات بروزرسانی را پردازش کند (متد execute()) و رویه‌هایی را همراه با پارامترهای آنها فراخوانی کند (کلاس CallableStatement).

پرسش‌های فصل

1- کدامیک از جملات زیر برای فراخوانی Stored Procedure با پارامترهای خروجی مناسب است؟

الف- Statement ب- PreparedStatement ج- CallableStatement

2- کدامیک از موارد زیر منجر به بارشدن درایور JDBC و ثبت آن توسط DriverManager نمی‌شود؟

الف- ;Class.forName(driverstring)

ب- ;()new DriverClass

ج- قراردادن نام درایور در خاصیت jdbc.drivers سیستم

د- هیچکدام

3- اطلاعات شبه‌داده با کدام شیء از DatabaseMetaData درخواست می‌شود؟

الف- Connection ب-ResultSet ج- DriverManager د- Driver

4- درصورت نیاز به ResultSet کدامیک از متدهای زیر را نمی‌توان استفاده کرد؟

الف- ()execute ب- ()executeQuery ج- ()executeUpdate

5- آیا ResultSet حاصل از متدی که یک statement را ایجاد و یک درخواست را اجرا می‌کند قابل اطمینان است؟

الف- بله ب- خیر

6- چگونه می‌توان از JDBC برای ایجاد یک بانک اطلاعاتی استفاده کرد؟
الف- اضافه‌کردن create=true در انتهای JDBC URL
ب- اجرای دستور "CREATE DATABASE jGurw"
ج- اجرای دستورات SQL "STRSQL" و "CREATE COLLECTION jGurw"
د- ایجاد بانک اطلاعاتی در DBMS خاص

7- کدامیک از کاراکترهای زیر برای نمایش پارامتر ورودی در CallableStatement استفاده می‌شود؟
الف- % ب- * ج- ? د- #

8- کدامیک از دستورات زیر نمی‌تواند اولین ستون از rs ResultSet ایجادشده پس از اجرای دستور "Select name, rank, serial from employee" را برگرداند؟
الف- rs.getString(0); ب- rs.getString("name");
ج- rs.getString(1);

9- کدامیک از موارد زیر با درایور بانک اطلاعاتی JDBC 2.0 انجام‌پذیر ولی با درایور JDBC 1.x انجام ناپذیر است؟
الف- ساختن Statement و فرستادن به بانک اطلاعاتی با همدیگر
ب- پیمایش در ResultSet به صورت مستقیم bi-directionally
ج- کار با انواع داده‌یی spl3 به صورت مستقیم
د- تمام موارد بالا

10- کدامیک از کلاس‌های زیر شامل متدهای کنترل تراکنش‌ها مانند setAutoCommit، Commit و rollback می‌باشد؟
الف- Connection ب- Statement ج- ResultSet