ACESSO À BASE DE DADOS ATRAVÉS DE ODBC E JDBC

Este capítulo é sobre programação de acesso à bases de dados. O objetivo principal é explicar como se programa o acesso à bases de dados, utilizando SQL através da linguagem Java.

No passado, nos defrontamos com uma dezena de produtos para bases de dados disponíveis, e cada um falando para nossas aplicações em sua própria linguagem. Se sua aplicação necessitasse falar a uma nova base de dados, você teria de ensiná-la (e a si próprio) essa nova linguagem. Por causa da confusão causada pela proliferação de APIs nativas de acesso às bases de dados proprietárias, surgiu a idéia de uma API de acesso universal às bases de dados.

A CLI SAG

A questão de se ter que aprender várias linguagens de APIs de diversos fornecedores foi resolvido através da criação de uma linguagem padrão única de acesso. Assim, historicamente, a comunidade da área de bancos de dados formou um consórcio, chamado SAG (**S**tandard Query Language **A**ccess **G**roup) constituído por vários fornecedores, para prover um padrão unificado de linguagem para acesso à base de dados remotas. Assim, nasceu SQL CLI (Standard Query Language Call Level Interface) definido pelo SAG.

Você deve notar que uma CLI não é uma nova linguagem de consulta. Ela é, simplesmente, uma interface procedural para SQL. Suas aplicações usam CLI para submeter *statements* SQL para um sistema gerenciador de base de dados (DBMS). Você usa SQL para realizar consultas e atualizações numa base de dados. Assim, você pode pensar em CLI, exatamente como um envólucro para SQL (SQL wrapper). Uma CLI nem adiciona, nem subtrai poder de SQL. CLI é, simplesmente, um mecanismo para submissão de *statements* SQL.

O CLI requer o uso de *drives* de base de dados que aceitam uma chamada CLI e traduzem essa chamada para a linguagem nativa de acesso aos servidores de base de dados. Com o *driver* apropriado, cada aplicação-fonte de dados pode funcionar como um servidor-CLI, e pode, então, ser acessada por ferramentas *front-end* e programas-clientes que usam a CLI. Uma CLI requer um *driver* para cada base de dados para qual ela se conecta. Cada *driver* deve ser escrito para um servidor específico, usando os métodos de acesso existentes para bases de dados. A CLI provê um *driver manager* que fala com um driver através de uma SPI (Service Provider Interface).

A CLI ODBC da Microsoft

A API padrão Windows *Open Database Connectivity* (ODBC) da Microsoft é uma versão estendida da CLI SAG. Em agosto de 1992, a Microsoft liberou o ODBC 1.0 SDK. ODBC foi desenvolvida para criar um único padrão de acesso à base de dados no ambiente

WINDOWS. A idéia de ODBC foi criar uma maneira comum de acesso usando SQL (*ODBC Driver Manager*), de forma que uma aplicação-cliente pudesse acessar bases de dados de diferentes fornecedores, como por exemplo, Oracle Database, SQL Server (Microsoft), DB2 (IBM), entre outras, através de seus *drivers* correspondentes. Veja a figura 1 que mostra as camadas de ODBC (Layers of ODBC). ODBC foi a resposta shrink-wrapped (um pacote restrito) para acesso à base de dados sob a plataforma WINDOWS. ODBC é uma API procedural. ODBC lida com chamadas `a API. Embora a indústria tenha aceito ODBC como o meio primário para acessar bases de dados em WINDOWS, este produto não traduz bem no mundo de Java, que é um mundo orientado a objeto. ODBC é uma interface em C; ela não pode usada "com é" em Java.

ODBC 1.0 era lenta e "buggy"; ela foi limitada à plataforma WINDOWS e faltava documentação e exemplos de código necessários para elucidar dúvidas de desenvolvedores. Em abril de 1994, a Microsoft lançou o ODBC 2.0 SDK, que terminou com muitos dos problemas com o *driver manager* anterior. Em dezembro de 1994, os primeiro drives de 32-bits ODBC 2.0 foram encomendados.

ODBC existe para prover uma camada de abstração para linguagens como C e C++, bem como, ferramentas de desenvolvimento populares como DELPHI, POWER BUILDER e VISUAL BASIC. Mas, ODBC não provê a independência de plataforma do sistema operacional como Java.

ODBC 3.0, surgido em novembro de 1996, adiciona em torno de 300 páginas à especificação existente, introduz 20 novas chamadas de funções e suporta Unicode. ODBC 3.0 retorna mais informação, sobre aproximadamente 40 novos ítens. A Microsoft controla totalmente o padrão. A pergunta que se faz ainda hoje, é: O futuro de ODBC se aliará com SQL3/CLI ou tornar-se-á um padrão proprietário baseado em OLE ? Microsoft promete suporte à SQL3 e ainda anuncia OLE/DB como o produto futuro para substituir ODBC.

ODBC tem muitas desvantagens. A mais séria é que é uma especificação controlada pela Microsoft, e está constantemente sendo estendido. Seu futuro é também incerto, dado o compromisso corrente da Microsoft com o OLE/DB, que introduz um paradigma diferente de programação – é baseado em objeto e não procedural. Os *drivers* atuais têm diferentes níveis de compatibilidade, que não são bem documentados. As camadas ODBC introduzem uma porção de *overhead* (especialmente para atualizações e inserções SQL), e não são nunca tão rápidos quanto as APIs nativas. Para funções simples de leitura-somente, os drivers ODBC estão agora em torno de 10% da performance de um driver nativo.

A X/Open SAG CLI

X/Open é uma organização independente, voltada para sistemas abertos. Sua estratégia é combinar vários padrões dentro de um ambiente de sistemas integrados compreensivo, chamado CAE (*Common Application Environment*), que correntemente contém um *portfolio* envolvente de APIs práticas. X/Open suporta suas especificações com um conjunto extensivo de testes de conformidade que um produto deve passar para obter marca registrada X/Open (o XPG brand).

As APIs SAG são baseadas sobre **SQL dinâmico** . Elas permitem você se conectar à base de dados através de um *driver* local (3 chamadas), preparar requisições SQL (5 chamadas), executar as requisições (2 chamadas), terminar um SQL statement (3 chamadas) e terminar uma conexão (3 chamadas).

Em dezembro de 1994, SAG voltou esta simples API para X/Open; ela é agora chamada **X/Open CLI** para diferenciar de ODBC da Microsoft e outras limitações SAG. No início de 1996, **X/Open CLI** tornou-se um padrão internacional ISO 9075-3 "Call Level Interface". **X/Open SQL CLI** é atualmente a base para a interface ODBC da Microsoft. O futuro **SQL3 CLI** é baseada, também, sobre **X/Open CLI**, com algumas extensões.

X/Open SQL CLI e **ODBC** – incluem drivers, conexões, SQL statements, cursores, transações, conjuntos de resultados, procedures armazenadas, exceções, entre outros elementos.

CLI Versus Embedded SQL

Uma alternativa à CLI é Embeddeed SQL (ESQL), que é o padrão definido ISO SQL-92, para embutir statements SQL "como é" dentro de linguagens de programação ordinárias. O SQL-92 especifica a sintaxe para embutir SQL em C, COBOL, FORTRAN, PL/1, PASCAL, MUMPS e ADA. Cada statement SQL é "flagged" com identificadores específicos da linguagem, que marcam o início e o fimde uma statement SQL. Esta abordagem requer rodar SQL através de um pré-compilador para gerar um arquivo de código-fonte que o compilador da linguagem entenda.

Da perspectiva de um pacote cliente-servidor, o maior obstáculo com ESQL é que a base de dados alvo deve ser conhecida e estar disponível quando o programa está sendo desenvolvido. Isto torna difícil ligar um programa-cliente à uma base de dados em tempo de execução.

Em geral, o que se tem hoje em termos de SQL CLIs é que essas são flexíveis, mas lentas. Sistemas de alta performance requerem o uso **procedures armazenadas** que executam as statements SQL. Bases de dados proporcionam dois tipos de Prepared SQL: **statements preparadas** e **procedures armazenadas**; elas diferem somente, no que podemos enviar a statement SQL à base de dados para ser interpretada antes de você realmente usá-la em sua aplicação, ou no caso em que as statements SQL já estão prontas para serem executadas através de procedures previamente situadas junto à uma base de dados. CLIs – como ODBC e JDBC – tipicamente, permitem invocar uma procedure armazenada de um fornecedor específico usando *pass-throughs* (você especifica o nome da procedure armazenada e o servidor, como parâmetros na chamada *execute*). X/Open, eventualmente, especificará uma chamada de procedure armazenada CLI, baseada em SQL3. Em todos os casos, CLI é ainda demais lenta quando comparada à performance de APIs nativas.

O que é JDBC (Java DataBase Connection)

Trabalhando com as empresas líderes no campo de bases de dados (Sybase, Informix, IBM e outras), JavaSoft desenvolveu um API única para acesso à base de dados. O objetivo foi prover uma interface orientada a objeto, para acessar uma base de dados relacional. Como parte desse processo, eles guardaram três principais metas em mente:

- JDBC deveria ser uma API a nível de SQL. Bases de Dados SQL é o modelo de aplicação que mais aparece como cliente/servidor. Uma API a nível de SQL significa que JDBC permite-nos construir *statements* SQL e embutí-las dentro de chamadas de uma API Java.

- JDBC deveria capitalizar a experiência de APIs de bases de dados existentes. De fato JavaSoft projetou sobre os aspectos de sucesso da API de ODBC.

- JDBC deveria ser simples. JDBC tenta ser tão simples quanto possível, enquanto provendo desenvolvedores com a máxima flexibilidade.

JDBC é um conjunto de classes e interfaces em Java, que proporcionam uma interface similar a ODBC para bases de dados SQL.

Você pode usar JDBC, de dentro de seus programas Java, para acessar quase todos as bases de dados SQL, incluindo ORACLE 7, SYBASE, DB2, SQL SERVER, ACCESS, FOXBASE, PARADOX, PROGRESS, SQLBase e XDR. Drivers JDBC estão disponíveis em Symantec, Intersolv, IBM, JavaSoft, e Inprise (Borland e Visigenic).

Classes JDBC para criar Conexão

`java.sql.DriverManager`

`DriverManager` é uma classe. Sua principal função é manter uma lista de *drivers* de diferentes fornecedores e associar uma aplicação-cliente com um *driver* que corresponda ao URL requerido.

`java.sql.Connection`

A classe `Connection` representa uma única transação da base dados. Você usa `Connection` para enviar uma serie de *statements* SQL à base de dados e gerenciar a realização completa da transação (*commiting*) ou o aborto dos *statements* da mesma (*aborting*).

java.sql.SQLException

A classe `SQLException` herda da classe geral `java.lang.Exception`, que provê informação extra sobre erros nos acessos à base de dados.

Se você obtém diversos erros durante a execução de uma transação, você pode encadeá-los todos eles juntos nesta classe. Isto é frequentemente útil quando você tem exceções que você deseja permitir que o usuário saiba, mas você não deseja parar o processamento.

Um exemplo de uma simples conexão é obtida com o seguinte código JDBC:

```
inport java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

/**
* Esta classe jdbc_Connection é um programa que aceita a
* seguinte linha de comando:
* JDBC SimpleConnection  DRIVER  URL  UID  PASSWORD
* Se a URL corresponde ao "site" do driver especificado, então este * será carregado e uma conexão deverá obtida.
*/

public class   jdbc_Connection {
    static public void main (String args [ ] )  {

        //  Declara um objeto da classe Connection ainda inexistente.
        Connection connection = null;

        //  Processa a linha de comando
        if  (  args.lengtht  !=  4  )  {
            System.out.println ( "JDBC SimpleConnection " +
                        "DRIVER  URL  UID  PASWORD" );
            System.exit (0);
         }

         //  carrega a ponte-driver jdbc:odbc

        try  {
                Class.forName ( "jdbc.odbc.JdbcOdbcDriver" );
         }
         catch ( Exception e )  { // Falha ao carregar o driver
             System.err.println ( "Falha ao carregar o driver" );
             System.exit (1);
         }

         String url =   "jdbc:odbc://150.162.63.7/DB_Teste";
         try  {
```

```
            // entra o nome do usuário (user name) em args [0],
            //  neste nosso caso, você só entra com o seu nome.

          // obtém a conexão ...
          System.out.println ( "Conectando ao URL especificado" );
          connection = DriverManager.getconnection ( url,
                                  args [2],  "  " );
          System.out.println ( "Conexão bem sucedida !" );
        }
        catch  ( SQLException e )  {
            e.printStackTrace();
        }

   }
}
```

Fechamento de uma Conexão: o Método Close()

As classes `Connection`, `Statement`, e `Result` têm o método `close`(), para fechamento da conexão. Uma dada implementação JDBC permitirá ou não, requerer o fechamento da conexão antes que alguns objetos sejam reutilizados. Pode acontecer de existir um objeto que seja bom não ser destruído. Ele poderá estar retendo recursos importantes da base de dados. É, portanto, sempre uma boa idéia fechar qualquer instância desses objetos, quando você já tiver feito tudo com eles.

Gerando Instruções de Atualização

Um determinado grupo de instruções SQL pode ser considerado como de atualização, no sentido de que não se deve esperar que retornem linhas de tabelas. Se encaizam neste gruopo as instruções de CREATE TABLE, INSERT, UPDATE e DELETE.

JDBC fornece um método para enviar instruções de atualização sem esperar por resultados. O objeto Statement, que é fornecido pelo objeto Connection, provê o método executeUpDate (). Para executaá-lo é preciso obter antes um objeto Statement. Depois de feito uma conexão bem sucedida, você pode recuperar um objeto Statement com uma linha:

```
Statement  stmt  =  connection.createStatement ();
```

Então, você pode executar instruções de atualização:

```
stmt.executeUpDate ( "CREATE TABLE  .... " );
stmt.executeUpDate ( "INSERT INTO ...  " );
```

Quando terminar de usar o objeto Satement, ele deve ser fechado como você faz com uma conexão.

stmt.close ();

Para criar uma tabela, o seguinte código precisa ser executado:

Exemplo 2:

```
stmt.executeUpDate ( " CREATE TABLE  DB_Teste
  ( id_prod int,  nome_prod CHAR (10), quantidade  int,
                  preço  float,  taxa_icms  float ) "  );
```

Após a execução deste comando, as linhas na tabela devem ser inseridas através de comandos como:

```
stmt.executeUpDate ( "INSERT INTO   DB_Teste  ( nome_coluna,
nome_coluna, ....  )  VALUES  (  valor_coluna, valor_coluna, ....  );
```

Acesso básico à Base de Dados

Agora que você está conectado à base da dados, você pode começar a fazer atualizações e consultas.

O tipo mais básico de acesso à base de dados envolve escrever código JDBC para *statements* de atualizações (INSERT, UPDATE, DELETE), ou consultas (SELECT).

O acesso básico inicia com o objeto `Connection` que você criaou antes. Quando este objeto é criado, ele é uma simples ligação direta à base de dados. Você usa um objeto `Connection` para gerar implementações de `java.sql.statements` vinculadas a uma mesma transição.

Após você ter usado uma ou mais objetos da classe Statement gerados por sua Connection, você pode usar ela para fazer *commit* ou *rollback* com os objetos *statements* associados com a `Connection`.

Um objeto `Statement` é uma declaração SQL. Uma vez que você tenha obtido um objeto `Connection`, você tem os campos de como se fosse um "cheque em branco" que você pode preencher para a transação representada pelo objeto `Connection`.

JDBC usa um método diferente para enviar consultas, em relação a atualizações. A diferença chave é o fato que o método para consultas retorna uma instância de `java.sql.Result`, enquanto o método de não-consultas retorna um inteiro. A classe `Result` provê você com acesso aos dados recuperados pela consulta.

Classes de Acesso Básico à Bases de Dados JDBC

No que segue são explicadas as classes para acesso básico a uma base de dados usando JDBC.

java.sql.Statement

A classe `Statement` é a mais básica das três classes JDBC representando *statements* SQL. Ela realiza todas as *statements* SQL básicas. Em geral, uma simples transação usa somente um dos três métodos de execução de *statements* na classe `Statement`. O primeiro método, `executeQuery`(), toma um *string* SQL como um argumento e retorna um objeto da classe `Result`. Este método deve ser usado para qualquer chamada SQL que espera retornar dados da base de dados. *Statements* "`UpDate`" são executadas usando o método `executeUpDate`(). Este método retorna o número de linhas afetadas na base de dados.

`java.sql.Result`

`Result` é uma classe da qual um objeto é instanciado, representando uma linha de dados retornada por uma consulta à base de dados.

A classe `Result` provê uma série de métodos para recuperar colunas dos resultados de uma consulta à base de dados. Os métodos para obtenção de uma coluna toda toma a forma:

type get type (int | string)

onde os argumentos representam ou o número da coluna ou o nome da mesma.

Porque a classe `Result` manipula uma única linha em qualquer dado tempo, a classe provê o método `next`() para se fazer referência para a próxima linha de um conjunto de resultados.

Modificando a Base de Dados

No exemplo anterior, nós usamos JDBC para realizar uma simples conexão. É claro que, não podemos realmente recuperar dados de uma base de dados antes de termos colocado dados lá.

O Exemplo 3 mostra uma classe de atualização, usando SQL INSERT, suprida com a implementação JDBC para o *driver* associado ao seu fornecedor de banco de dados. No caso, usaremos o *driver* ODBC do Microsoft Access.

Emitindo Instruções de Atualização com SQL Insert

Exemplo 3:

```
class  jdbc_Insert  extends  jdbc_Connection  {

    public static void main (String args [])  {

        if ( args.length !=  5 )  {
           System.out.println ( "JDBC Inserindo" + "ID_PROD
                        NOME_PROD  QTD  PREÇO  TAXA" );
        System.exit (0);
        }
        try  {

          //  Obtém dados pela linha de comando.
           String id_prod = args [0];
           String nome_prod = args [1];
           String quantidade = args [2];
           String preço =  args [3];
           String taxa_icms  =  args [4];

          // Obtém um objeto Statement de Connection.
           Statement stmt  = connection.createStatement();

           //  Insere uma linha em DB_Teste.
           stmt.executeUpDate (  "INSERT
              INTO  DB_Teste  ( id_prod, nome_prod, quantidade,
                                        preço, taxa_icms )
               VALUES (" + id_prod + ",  " + nome_prod +",
                                 " + quantidade +",  " + preço +",
                                 " + taxa_icms +" ) "   );
           stmt.close();
        }
         catch  (Exception e )  {
            e.printStackTrace();
        }
    }
}
```

Observe que, fazer uma chamada uma chamada à base de dados, não é nada mais que criar um `Statement` e passar a declaração SQL via um de seus métodos `execute`. Diferente de `executeQuery`(), contudo, `executeUpDate`() não retorna um `ResultSet` (você não deve estar esperando quaisquer resultados). Ao invés disso, ele retorna o número de linhas afetadas pelas declarações SQL, UPDATE, INSERT ou DELETE.

Por *default*, JDBC realiza (commits) cada *statement* SQL quando ela é enviada à base de dados; isto é chamado *autocommit*. Contudo, uma manipulação mais robusta de erros, você pode estabelecer um objeto `Connection` que emite uma série de mudanças que não têm nenhum efeito sobre a base de dados, até que você expressamente envie um *commit*. Cada `Connection` é separada, e um commit sobre ela não tem nenhum efeito sobre o statement da outra. A classe `Connection` provê o método `setAutoComit`(), assim você pode tornar o *autocommit* desligado (off). O exemplo 3 mostra uma simples aplicação que torna o *autocommit* desligado e realiza (*commits*) duas *statements* juntas ou não, sendo o último caso correspondendo ao caso em que a transação não consegue ser realizada.

Emitindo Instrução de Atualização com SQL UPDATE

Pode ser muito útil definir o número de linhas afetadas por uma instrução de atualização. O objeto Statement fornece um método, getUpDateCount () para fazer exatamente isso. Retorna um valor inteiro depois de qualquer tipo de operação de atualização que possa ser utilizado para determinar se algumas linhas foram afetadas pela operação ou não, e quantas linhas foram afetadas.

Mostramos no Exemplo 4, uma classe JDBC com instrução de atualização UPDATE, que verifica e conta as linhas afetadas.

Exemplo 4:

```
class  jdbc_UpDate  extends  jdbc_Connection {
    public static void main (String args [])  {


        if ( args.length !=  5 )  {
           System.err.println ( “Erro, JDBC Inserindo” + “Entre com
               ID_PROD   NOME_PROD   QTD   PREÇO  TAXA” );
           System.exit(1);
        }
        try  {
           //  Obtém um objeto Statement de Connection
           Statement stmt = connection.createStatement ();

           //  Declara duas strings.
           String id_prod, nome_prod;

           //  Emite uma atualização simples.
           stmt.executeUpDate (  “UPDATE
                   <nome_DB>  SET quantidade =  args[2],
                   preço = args[3]   WHERE id_prod = args[0]
                   AND nome_prod  =  args[1]”  );
```

```
        //  Obtém o valor do contador de atualizações.
          int  update_count  =   stmt.getUpDateCount();
          System.out.println (update_count + "linhas atualizadas." );

        //   Fecha a statement SQL criada para UpDate.
          stmt.close();

          //   Fecha a conexão estabelecida com a base de dados.
          connection.close();
        }
        catch  (Exception e )  {
          System.out.println (e.getMessage() );
          e.printStackTrace();
        }
    }
}
```

Emitindo uma Instrução de Atualização com SQL DELETE

No exemplo 5, a seguir, está o código JDBC para se realizar uma statement SQL DELETE:

```
class  jdbc_Delete  extends  jdbc_Connection {

    public static void main (String args [])  {

        if ( args.length !=  5 )  {
          System.out.println ( "JDBC Deletando" + "ID_PROD
                    NOME_PROD  QTD  PREÇO  TAXA_ICMS" );
          return;
        }
        try  {
          Statement stmt = connection.createStatement();
          String nome_prod;
          int  update_count  =  stmt.executeUpDate (  "DELETE
                  FROM  <nome_DB>"  +
                   " WHERE  nome_prod =  args [1] "   );
           System.out.println ( update_count  + "linhas deletadas." );
           stmt.close();
           connection.close ();
        }
        catch  (Exception e )  {
          e.printStackTrace();
        }
    }
}
```

Consultando à Base de Dados

O exemplo 6 mostra o código para a lógica de uma simples consulta SELECT da implementação JDBC para a ponte de drivers JDBC:ODBC. A classe jdbc_Select se comunica com o `DriverManager` JDBC para ligá-la à implementação adequada da base de dados, situada na URL indicada. O objeto Statement criado, realiza a consulta SELECT. Um objeto `ResultSet` então provê à aplicação-cliente com os campos `key` e `val` da tabela `t_test`.

Executando Consultas com SELECT e Recuperando os Resultados

Com dados armazenados em DB_Teste, podemos agora mostra um exemplo em que é feita uma consulta sobre o preço e a quantidade com relação a um produto determinado dentro do statemente SQL SELECT. O método utilizado para realizar uma consulta é `executeQuery()`, o qual retorna um objeto `ResultSet.`

Para buscar a primeira linha, de acrdo com a consulta,você deve chamar o método next () de conjunto de resultados:

```
result.next ();
```

Se houver mais linhas, de acordo com a consulta, pode ser utilizado o next () para buscá-las e prepará-las para serem recuperadas com um dos métodos get* (). Desde que next () responda com um valor booleano e não lance nenhuma execeção quando não há mais linhas, é possível utilizar next () com segurança para testar a existência de mais linhas em um loop while ou situação similar. Uma vez que você tenha utilizado next () para buscar pelo menos uma linha, poderá utilizar um dos métodos get* (), tal como getString () ou getInt ():

```
String  xxxx  =  result.getString ( “xxxx” );
Int  yyyy  =  result.getInt ( “yyyy” );
```

Exemplo 6:

```
public class  jdbc_Select  extends  jdbc_Connection  {

   public static void main ( String args[] )  {

      try  {
         // Obtém um objeto Statement a partir de Connection.
         Statement select  =  connection.createStatement();

         //  Executa Consulta e retorna um conjunto de resultados.
         ResultSet  result  =  select.executeQuery ( “ SELECT
                                    preço, quantidade
```

```
                                FROM  <nome_db>
                                WHERE  nome_prod  =  args[1] ”  );
            //
            //  Obtendo os resultados.
            System.outprintln ( “Obtendo o Conjunto de Resultados: ” );
            //
             while ( result.next () )
                {
                  float preço  =   result.getFloat ( “preço” );
                  int quantidade =   result.getInt ( “quantidade” );

                  //  Mostrando o conjunto de resultados.
                  System.out.println ( “preço = ”  +  preço );
                  System.out.println ( “quantidade =” + quantidade );
                }

            //  Fecha instrução de consulta.
            stmt.close();

            //  Fecha a conexão com DB_Teste.
            System.outprintln ( “Fechando a Conexão” );
            connection.close();
        }
        catch  (Exception e )  {
             System.out.println (e.getMessage () );
             e.printStackTrace();
        }
    }
}
```