

Índice

JDBC	2
Introdução	2
Conceito	2
Programação Básica JDBC.....	3
Carregamento do “Driver”	4
Criação de conexão	4
Criação de Comando	4
Execução de comando.....	4
Iteração sobre os resultados.....	5
Programa de SQL dinâmica.....	5
Utilização de PreparedStatement.....	8
Atualização da posição do cursor.....	12
Modos de Transação.....	15
Aplicação CGI.....	19
Acesso a Metadados	24
Percurso sobre o “array” ResultSet	28
Exemplo de Percurso sobre o “array” ResultSet	33
“Applets” JDBC	38
Exemplo de “Applet JDBC”	38
Código de um “Applet”	39
Declarações	39
Exibição da janela do “Applet”	39
Declaração da Classe Button	39
Manipulador de acionamento do botão Previous	39
Manipulador de acionamento de botão InsertData	40
Evento de Botão NextRow	40
Evento de Botão GetData	40
Componentes e Java Beans	41
Introdução	41
Exemplo de Java Bean : Sinal de Trânsito	43
Projeto Ponto de Vendas	46
Generalidades	46
Informação Explícita através de BeanInfo	49
Juntando tudo	51
Consulta de preços.....	55
Trabalhando com Imagens	57

JDBC

Introdução

Conceito

O objetivo dos criadores de JDBC foi desenvolver uma API de baixo nível que suportasse acesso SQL. Um produto similar é o ODBC. Mas ODBC é um padrão para a linguagem C. Java requer um padrão que possa ser suportado em Java. JDBC representa uma coleção de classes orientadas a objeto, em um “**package**” Java, que pode ser diretamente associado às funções ODBC similares. JDBC representa a funcionalidade ODBC através de um “conjunto de classes” e não simplesmente através de uma “coleção de funções”, como ODBC faz.

O emprego básico de JDBC compreende as seguintes etapas:

- Carregamento de um “driver” de BD
- Criação de um objeto **Connection**
- Criação de um objeto **Statement**
- Execução de um comando SQL com o objeto **Statement** retomando um objeto **ResultSet**
- Recuperação de linhas de dados utilizando o objeto **ResultSet**

No caso de “applets”, os métodos JDBC são usualmente chamados por eventos disparados por botões e destinam-se a recuperar dados e exibi-los na janela do “applet”.

Uma aplicação bastante comum nas páginas WWW é uma aplicação CGI para recuperar dados de um BD. A aplicação recebe um conjunto de parâmetros, analisa esses parâmetros e retorna dados formatados em páginas HTML.

O processamento de consultas SQL compreende sua interpretação e otimização antes da execução. Estas tarefas podem ser evitadas durante cada execução de consulta SQL utilizando comandos preparados (“prepared statements”, da classe **PreparedStatement**) que entram imediatamente em execução.

A recuperação de dados pode ser feita por arquiteturas cliente servidor em duas camadas ou em três camadas. A arquitetura em duas camadas é limitada quando o número de linhas a ser recuperado é muito grande ou quando existem restrições de segurança. Quando se utiliza uma arquitetura em três camadas a camada intermediária recebe as requisições por uma conexão convencional via “sockets” ou pela utilização de uma “remote method invocation”, ou RMI.

As aplicações CGI estão presentes em toda a WWW. Os “applets” usando JDBC podem eliminar a maioria dos programas CGI, mas as restrições de segurança e desempenho podem manter atraentes programas CGI, que podem ser escritos em Java.

Quando uma aplicação necessita de informações sobre o BD ao qual está conectada ela pode utilizar métodos JDBC que implementam funções de metadados e obter essas informações.

Aplicações JDBC

As aplicações consideradas como exemplos típicos de acesso a Bancos de Dados na Internet são as seguintes:

1. Exibição do conteúdo de uma tabela sendo o comando SQL e a tabela estáticos.
2. Exibição do conteúdo de uma tabela sendo o comando SQL estático e a tabela dinâmica.
3. Utilização de comandos pré compilados **PreparedStatement**.
4. Movimentação não seqüencial do cursor em **ResultSet**.
5. Utilização de meta dados.
6. Utilização do conceito de transações em SQL.
7. Atualização de tabelas.
8. Aplicações com CGI.
9. Utilização de Applets.
10. Aplicações em três camadas com invocação remota de métodos (RMI).

No Curso de Bancos de Dados para Internet serão apresentados exemplos das oito primeiros tipos de aplicações, ficando os dois últimos para o Curso de Java.

Programação Básica JDBC

A aplicação típica de JDBC compreende uma seqüência de chamadas como a que se segue:

- Carregamento de “driver”
- Criação de uma conexão
- Criação de um comando
- Execução do comando e retorno de um **ResultSet**
- Iterações sobre o **ResultSet** se ele existir

As aplicações JDBC normalmente envolvem uma combinação destas chamadas com chamadas a métodos de metadados e métodos de controle de transações. Os resultados das consultas são retornados em objetos **ResultSet**, o equivalente JDBC ao cursor (de SQL embutida). Um objeto metadados do **ResultSet** fornece informações tais como seu número de colunas, o tipo de dados, o tamanho e a precisão das colunas.

Uma variável **rs** que seja uma instância da classe **ResultSet** contém as linhas resultado de uma consulta SQL. Cada linha é composta de colunas, numeradas desde 1 até **rs.getColumnCount**. A recuperação dos dados contidos em cada coluna é feita empregando os métodos **getXXX** do tipo **apropriado** (**getBytes**, **getShort**, **getLong**, **getFloat**, **getDouble**, **getBigDecimal**, **getBoolean**, **getString** e outros). O parâmetro único dos métodos **getXXX** é o identificador da coluna desejada. Este identificador pode ser o nome da coluna (ente aspas) ou o seu número.

Meta dados são informações a respeito dos dados, ou seja, de como são organizados os dados. Os meta dados são tratados, em JDBC, por duas interfaces **DatabaseMetadata** e **ResultSetMetadata**. A classe **DatabaseMetadata** fornece informações sobre o banco de dados, como um todo, disponibilizando métodos para descobrir o que se pode fazer com muja dada combinação de BD e "driver". A classe **ResultSetMetadata** é usada para encontrar tipos e propriedades das colunas de um **ResultSet**.

Para a obtenção da lista de tabelas disponíveis em um BD utiliza-se o método **getTables** e, a seguir, o método **getColumns**, ambos de **DatabaseMetadata**. O resultado da aplicação deste último método gera um **ResultSet** colunas, no qual, podem ser extraídos os atributos, da forma:

```
String tipoDeDado = colunas.getString("TYPE_NAME");
String nomeColuna = colunas.getString("COLUMN_NAME");
int tamanhoDado = colunas.getInt("COLUMN_SIZE");
int digitos = colunas.getInt("DECIMAL_DIGITS");
int anulavel = colunas.getInt("NULLABLE");
boolean eNulo = (anulavel == 1);
```

Carregamento do “Driver”

O carregamento do “driver” normalmente é feito pelo método estático **forName** do objeto **Class**, como, por exemplo:

```
Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

Criação de conexão

Um “driver” JDBC não faz conexão com um BD, apenas criando um ambiente de programação. Uma conexão a BD é especificada através de sua **URL**, da sigla e da senha de acesso ao BD. Por sua vez a **URL** é composta de **jdbc:sub protocolo:sub nome**.

Sub protocolo é uma fonte de dados ou um mecanismo de conectividade a um BD que diversos “drivers” podem suportar, tais como ODBC, Oracle, ou outra biblioteca de acesso criada por algum usuário, por exemplo.

Sub nome é dependente de sub protocolo e é da forma

```
[/<nome de host>:<porta>/] sub sub nome
```

Sub sub nome é o nome do BD.

A conexão é feita por uma chamada ao método **getConnection** do **DriverManager**, da forma:

```
String url = "jdbc:odbc:msaccessdb";  
Connection con = DriverManager.getConnection (url, "", "");
```

Criação de Comando

A criação de comandos SQL é feita por chamadas ao método **createStatement** da classe **Connection**, da forma:

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

Os cursores **ResultSet** fazem parte da classe **ResultSet** e não da classe **Statement**.

Execução de comando

A execução dos comandos SQL é determinada por chamadas ao método **executeQuery**, no caso de consultas e ao método **executeUpdate**, no caso de atualizações, da forma:

```
String qs = "select * from funcionario";  
ResultSet rs = stmt.executeQuery( qs );
```

Ou

```
String criarTabela = "create table quadro" +  
"(nome_quadro varchar(20), id_quadro integer, valor float, " +  
"vendas integer, total integer)";  
ResultSet rs = stmt.executeUpdate ( criarTabela );
```

Iteração sobre os resultados

A classe **ResultSet** contém métodos que podem ser usados para iteração sobre os resultados de maneira seqüencial. Uma variável que seja uma instância de um **ResultSet**, **rs**, por exemplo, contém linhas resultantes de uma consulta SQL. Cada linha é composta de várias colunas, numeradas de 1 até **rs.getColumnCount()**. A recuperação dos valores contidos em cada coluna é feita usando o método **getXXX** do tipo apropriado, que pode ser **getBytes**, **getShort**, **getInt**, **getLong**, **getFloat**, **getDouble**, **getBigDecimal**, **getBoolean**, **getString** e outros. O único parâmetro de **getXXX** é o identificador da coluna desejada. Este identificador pode ser o nome da coluna (entre aspas) ou o número da coluna. Inicialmente faz-se uma chamada ao método **next** para posicionar o ponteiro para o primeiro elemento do **ResultSet**.

```
boolean more = rs.next();
```

A seguir um ciclo **while** percorre os resultados do **ResultSet**. O ciclo é controlado pela variável boolean **more**.

```
while ( more ) {
System.out.println( "Col1: " + rs.getInt( 1 ) );
more = rs.next();
}
```

Exemplo de Programa para exibição do conteúdo de uma tabela sendo o comando SQL e a tabela estáticos : Select1

Introdução:

Para executar um programa com SQL estática tudo é conhecido, exceto os valores dos atributos das linhas da tabela. O que vai ocorrer é apenas a exibição dos valores de atributos selecionados pela consulta.

Enunciado:

Exibir o conteúdo da coluna 1 de todas as linhas da tabela **funcionario**. A exibição de registros deve ser com uma linha por atributo de cada registro, da forma

Col 1: <Valor do atributo>

A url é **jdbc:odbc:msaccessdb**. A conexão deve ser obtida com **getConnection url,"Carla","Dalboni"**

A listagem completa é exibida a seguir.

```
import java.sql.*;
import java.io.*;

class Select1 {

public static void main( String argv[] ) {

    try {

        // Carga do "driver"
        Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );

        // Estabelecimento da conexão

        String url = "jdbc:odbc:msaccessdb";

        Connection con = DriverManager.getConnection (url,
            Carla", "Dalboni");

        // Geração da consulta
        String qs = "select * from funcionario";
        Statement stmt = con.createStatement( );
```

```

// Execução da consulta
ResultSet rs = stmt.executeQuery( qs );

// Percurso sobre o ResultSet imprimindo valor de coluna
boolean more = rs.next();
while ( more ) {
    System.out.println( "Coll: " + rs.getInt( 1 ) );

    // pode-se também retornar um inteiro como um string
    // System.out.println("Coll: " + rs.getString(1));
    more = rs.next();
}

}

catch ( java.lang.Exception ex ) {

// Impressão da descrição de exceção
System.out.println( "*** Erro na seleção de dados ** " );
ex.printStackTrace ();

}

}

}

```

Programa de SQL dinâmica

Uma aplicação mais elaborada aceita um argumento de linha de comando como, por exemplo, o nome da tabela a consultar. As consultas são construídas em tempo de execução e, para tanto, é necessário recuperar os metadados do BD para poder fazer essa construção.

Exemplo de Programa para exibição do conteúdo de uma tabela sendo o comando SQL e a tabela dinâmica : SelectGen

Introdução:

Para executar um programa com SQL dinâmica um programa pode aceitar argumentos da linha de comando, como por exemplo, um nome de tabela para a qual direcionar a consulta. Quando é empregado um “array” de argumentos na linha de comando o “array” é representado por um string que deve ser percorrido para recuperar os argumentos.

Como a consulta é montada em tempo de execução o número e os nomes das colunas do **ResultSet** são desconhecidos por ocasião da compilação. Esta informação pode ser obtida recuperando os meta dados do **ResultSet** usando o objeto **ResultSetMetaData** do **ResultSet** retornado pela consulta.

Enunciado:

Exibir o conteúdo de uma tabela. O programa deve receber o nome da tabela como argumento de linha de comando. Caso não seja fornecido este argumento será usada a tabela “default” **Funcionários**. A exibição de registros deve ser com uma linha por atributo de cada registro, da forma

Col <número da coluna> **Nome:** <Nome da coluna> **Valor :** <Valor do atributo>

A url é **jdbc:odbc:msaccessdb**. A conexão deve ser obtida com **getConnection url,"Carla","Dalboni"**

A carga do “driver” de BD e a conexão podem ser feitos da forma:

```

Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );
String url = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection (url, "", "");

```

A recuperação do nome da tabela do argumento da linha de comando é feita como:

```
String tableName = "funcionario";
if ( argv.length > 0 )
    tableName = argv[0];
```

O comando SQL **select** é construído como:

```
String qs = "select * from " + tableName;
```

A criação do comando e sua execução são feitos como:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery( qs );
```

Automaticamente cria-se o objeto **ResultSetMetaData**. Os dados podem ser obtidos como:

```
ResultSetMetaData rsmd = rs.getMetaData();
```

A travessia do conjunto de resultados é feita como:

```
int n = 0;
boolean more = rs.next();
while ( more ) {
    for ( n = 1; n <= rsmd.getColumnCount(); n++ ) {
        System.out.println( "Col " + n +
            " Nome: " + rsmd.getColumnName( n ) +
            " valor: " + rs.getString( n ) );
    }
    more = rs.next();
}
```

A listagem completa é exibida a seguir.

```
import java.sql.*;
import java.io.*;
import java.util.Date;

class SelectGen {

    public static void main( String argv[] ) {

try {

Date dt = new Date();
System.out.println( "A Date de hoje é " + dt.toString() );

// Carga do "driver"
Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );

// Estabelecimento da conexão
String url = "jdbc:odbc:msaccessdb";

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

// Definição da tabela da consulta
String tableName = "funcionario";
if ( argv.length > 0 )
    tableName = argv[0];

// Geração da consulta
String qs = "select * from " + tableName;
Statement stmt = con.createStatement();
```

```

// Execução da consulta
ResultSet rs = stmt.executeQuery( qs );

ResultSetMetaData rsmd = rs.getMetaData();

// Percurso sobre o ResultSet imprimindo nome e valor de colunas
int n = 0;
boolean more = rs.next();
while ( more ) {
    for ( n = 1; n <= rsmd.getColumnCount(); n++ ) {
        System.out.println( "Col " + n +
            " Nome: " + rsmd.getColumnName( n ) +
            " valor: " + rs.getString( n ) );
    }
    more = rs.next();
}

}

catch ( java.lang.Exception ex ) {

    // Impressão da descrição de exceção
    System.out.println( "*** Erro na seleção de dados *** " );
    ex.printStackTrace ();

}

}

}

```

Utilização de PreparedStatement

O processamento de cada consulta SQL apresentada ao motor de BD deve ser precedido de duas etapas:

- “Parsing” que compreende a verificação da correção sintática da consulta, da presença dos objetos de BD referenciados e de quais conversões de dados serão necessárias na exibição dos resultados.
- Otimização que compreende a escolha do melhor caminho de acesso para o processamento da consulta ou a ordem de execução das seleções e/ou junções. Esta otimização envolve operações tais como
 - Mover as operações **select** para que sejam executadas o mais cedo possível;
 - Combinar seqüências de operações **project**;
 - Eliminar operações redundantes simplificando expressões envolvendo relações vazias e predicados triviais;
 - Jamais recuperar uma tupla mais de uma vez;
 - Descartar valores desnecessários (não mais referenciados) de uma tupla logo depois da sua recuperação;
 - Construir relações com base no “princípio do menor crescimento”;
 - Otimizar a construção de junções.

Caso seja necessário repetir uma consulta uma grande número de vezes é mais conveniente executar as operações de “parsing” e otimização uma só vez e apenas substituir parâmetros nas partes das consultas que mudam a cada execução. Isto pode ser obtido usando a classe **PreparedStatement**. Usando esta classe um comando SQL pode ser montado com espaçadores (“place holders”) nos lugares aonde serão inseridos os parâmetros. Como caractere símbolo de espaçador usualmente emprega-se “?” e existem lugares específicos nas consultas nos quais estes espaçadores podem ser empregados. Durante a execução da consulta a atribuição de valores aos parâmetros é feita identificando a posição do parâmetro na consulta e fornecendo

seu valor, da forma:

setInt(int parameterIndex, int x) ou **setString(int parameterIndex, String x)**

Exemplo de utilização de comandos pré compilados PreparedStatements : PrepTest

Introdução:

O processamento de cada consulta SQL apresentada ao motor de BD deve ser precedido de duas etapas:

- “Parsing” que compreende a verificação da correção sintática da consulta, da presença dos objetos de BD referenciados e de quais conversões de dados serão necessárias na exibição dos resultados.
- Otimização que compreende a escolha do melhor caminho de acesso para o processamento da consulta ou a ordem de execução das seleções e/ou junções. Esta otimização envolve operações tais como
 - Mover as operações **select** para que sejam executadas o mais cedo possível;
 - Combinar seqüências de operações **project**;
 - Eliminar operações redundantes simplificando expressões envolvendo relações vazias e predicados triviais;
 - Jamais recuperar uma tupla mais de uma vez;
 - Descartar valores desnecessários (não mais referenciados) de uma tupla logo depois da sua recuperação;
 - Construir relações com base no “princípio do menor crescimento”;
 - Otimizar a construção de junções.

Caso seja necessário repetir uma consulta uma grande número de vezes é mais conveniente executar as operações de “parsing” e otimização uma só vez e apenas substituir parâmetros nas partes das consultas que mudam a cada execução. Isto pode ser obtido usando a classe **PreparedStatement**. Usando esta classe um comando SQL pode ser montado com espaçadores (“place holders”) nos lugares aonde serão inseridos os parâmetros. Como caractere símbolo de espaçador usualmente emprega-se “?” e existem lugares específicos nas consultas nos quais estes espaçadores podem ser empregados. Durante a execução da consulta a atribuição de valores aos parâmetros é feita identificando a posição do parâmetro na consulta e fornecendo seu valor, da forma:

setInt(int parameterIndex, int x) ou **setString(int parameterIndex, String x)**

Para mostrar a diferença de resultados que se pode obter utilizou-se, como exemplo, uma série de 1000 repetições das tarefas:

- Atribuir novo valor ao parâmetro da **PreparedStatement** (no caso o parâmetro inteiro é o índice da repetição)
 - Chamar o método **executeQuery**
 - Recuperar um novo **ResultSet** utilizando o mesmo objeto “container” que tinha sido usado anteriormente
- Os dados não foram exibidos pois isso não interessava ao processamento cuja finalidade principal consistia em executar operações consumidoras de tempo para medição de sua velocidade.

No exemplo a ser exibido usam-se duas versões da consulta. A versão com a classe **Statement** consome 6 segundos para 1000 iterações enquanto a versão usando a classe **PreparedStatement** efetua a mesma tarefa em 4 segundos.

Enunciado:

Utilizar um **PreparedStatement** para determinar o tempo de processamento de 1000 repetições da consulta

select nome from empregados where numero=?

A url é **jdbc:oracle:thin:@sbd:1521:orcl**. A conexão deve ser obtida com **getConnection url,"java01","java01"**. O “driver” a utilizar é **oracle.jdbc.driver.OracleDriver**.

- Carga do “driver” e criação da conexão

```
Class.forName ("oracle.jdbc.driver.OracleDriver");  
String url = "jdbc: oracle:thin:@sbd:1521:orcl ";
```

```

Connection con = DriverManager.getConnection(url,"java01","java01");
        ("jdbc:oracle:thin:@sbd:1521:orcl","java01","java01");
con.setAutoCommit(false);

```

- Criação do string de consulta com parâmetros e do objeto **PreparedStatement**

```

String qs = "SELECT NOME FROM EMPREGADOS WHERE NUMERO=?";
        // Prepara um comando SQL
PreparedStatement prepStmt = con.prepareStatement(qs);

```

- Estabelecimento do valor de parâmetro e execução de consulta

```

Calendar cal = Calendar.getInstance();
System.out.println("Inicio: "+cal.get(Calendar.HOUR_OF_DAY)+
        ":"+cal.get(Calendar.MINUTE)+
        ":"+cal.get(Calendar.SECOND));
long segini = cal.get(Calendar.SECOND);
int n = 1;
prepStmt.setInt( 1, n );
ResultSet rs = prepStmt.executeQuery();
boolean more = rs.next();

```

O comando `setInt(1, n)` atribui o valor `n` ao parâmetro 1 do **prepStmt**.

- Ciclo de 1000 iterações

```

        // Percorre o ResultSet
for ( ; n <= 1000 ; n++ ) {
    //System.out.println ("Nome = " + rs.getString("NOME"));
    prepStmt.setInt( 1, n );
    rs = prepStmt.executeQuery();
    more = rs.next();
}

```

```

cal = Calendar.getInstance();
System.out.println("Final: "+cal.get(Calendar.HOUR_OF_DAY)+
        ":"+cal.get(Calendar.MINUTE)+
        ":"+cal.get(Calendar.SECOND));
long segfim = cal.get(Calendar.SECOND);

// Exibe tempo esperado

long seconds = (segfim - segini);
System.out.println( "Tempo estimado: " + seconds +
        " segundos por " + n + " tuplas." );
}

```

- Comparção do processamento sem **PreparedStatement**

```

// Recomeçar sem PreparedStatement
cal = Calendar.getInstance();
System.out.println("Inicio: "+cal.get(Calendar.HOUR_OF_DAY)+
        ":"+cal.get(Calendar.MINUTE)+
        ":"+cal.get(Calendar.SECOND));

segini = cal.get(Calendar.SECOND);
n = 1;
Statement stmt1 = con.createStatement ();
String consulta ="SELECT NOME FROM EMPREGADOS WHERE NUMERO=";
ResultSet rs2 = stmt1.executeQuery(consulta + n);
        more=rs2.next();
        // Percorre o ResultSet

```

```

for (n=1; n <= 1000 ; n++ ) {
    rs2    = stmt1.executeQuery(consulta + n);
    more = rs2.next();
}

cal = Calendar.getInstance();

System.out.println("Final: "+cal.get(Calendar.HOUR_OF_DAY)+
    ":"+cal.get(Calendar.MINUTE)+
    ":"+cal.get(Calendar.SECOND));
segfim = cal.get(Calendar.SECOND);

    // Exibe tempo esperado

    seconds = (segfim - segini);
    System.out.println( "Tempo estimado: " + seconds +
        " segundos por " + n + " tuplas." );
con.commit();
}

```

A listagem completa é exibida a seguir.

```

import java.sql.*;
import java.io.*;
import java.util.Date;
import java.util.Calendar;

class PrepTest {

    public static void main( String argv[] ) {

        try {

            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@sbd:1521:orcl","java01","java01");
            con.setAutoCommit(false);

            String qs = "SELECT NOME FROM EMPREGADOS WHERE NUMERO=?";
            // Prepara um comando SQL
            PreparedStatement prepStmt = con.prepareStatement(qs);
            Calendar cal = Calendar.getInstance();
            System.out.println("Inicio: "+cal.get(Calendar.HOUR_OF_DAY)+
                ":"+cal.get(Calendar.MINUTE)+
                ":"+cal.get(Calendar.SECOND));
            long segini = cal.get(Calendar.SECOND);
            int n = 1;
            prepStmt.setInt( 1, n );
            ResultSet rs    = prepStmt.executeQuery();
            boolean more = rs.next();

            // Percorre o ResultSet
            for (; n <= 1000 ; n++ ) {
                //System.out.println ("Nome = " + rs.getString("NOME"));
                prepStmt.setInt( 1, n );
                rs    = prepStmt.executeQuery();
                more = rs.next();
            }
        }
    }
}

```

```

cal = Calendar.getInstance();
System.out.println("Final: "+cal.get(Calendar.HOUR_OF_DAY)+
    ":"+cal.get(Calendar.MINUTE)+
    ":"+cal.get(Calendar.SECOND));
    long segfim = cal.get(Calendar.SECOND);

// Exibe tempo esperado

long seconds = (segfim - segini);
System.out.println( "Tempo estimado: " + seconds +
    " segundos por " + n + " tuplas." );

    // Recomeçar sem PreparedStatement
cal = Calendar.getInstance();
System.out.println("Inicio: "+cal.get(Calendar.HOUR_OF_DAY)+
    ":"+cal.get(Calendar.MINUTE)+
    ":"+cal.get(Calendar.SECOND));

    segini = cal.get(Calendar.SECOND);
    n = 1;
Statement stmt1 = con.createStatement ();
String consulta ="SELECT NOME FROM EMPREGADOS WHERE NUMERO=";
ResultSet rs2 = stmt1.executeQuery(consulta + n);
    more=rs2.next();
        // Percorre o ResultSet
for (n=1; n <= 1000 ; n++ ) {
    //System.out.println( "Nome = " + rs2.getString("NOME") );
    rs2 = stmt1.executeQuery(consulta + n);
    more = rs2.next();
}

cal = Calendar.getInstance();

System.out.println("Final: "+cal.get(Calendar.HOUR_OF_DAY)+
    ":"+cal.get(Calendar.MINUTE)+
    ":"+cal.get(Calendar.SECOND));
    segfim = cal.get(Calendar.SECOND);

// Exibe tempo esperado

seconds = (segfim - segini);
System.out.println( "Tempo estimado: " + seconds +
    " segundos por " + n + " tuplas." );

con.commit();
}

catch (java.lang.Exception ex) {

// Tratamento de erros.
System.out.println( "*** Erro na seleção de dados ** " );
ex.printStackTrace ();

}
}
}

```

Atualização da posição do cursor

Uma aplicação relativamente freqüente consiste na leitura de dados com um cursor e na atualização de linhas de maneira seletiva baseada em informação obtida no processo de recuperação. A atualização seletiva pode ser feita por outro comando SQL para a busca e atualização do registro ou então pela atualização da linha corrente do cursor. A segunda opção é mais eficiente e será exibida a seguir:

- Carga do “driver” e criação da conexão

```
Class.forName ( "jdbc.odbc.JdbcOdbcDriver" );  
String url = "jdbc:odbc:informix5";  
Connection con = DriverManager.getConnection (url, "", "");
```

- Criação do objeto **DatabaseMetaData** e teste da capacidade de atualização da posição

```
// need a database that supports positioned updates  
DatabaseMetaData dmd = con.getMetaData();  
if ( dmd.supportsPositionedUpdate() == false ) {  
    System.out.println( "Positioned update is not supported by this  
database." );  
    System.exit( -1 );  
}
```

- Execução da consulta seletiva

```
Statement stmt1 = con.createStatement();
ResultSet rs = stmt1.executeQuery( "select " +
    " * from loadtest where coll = 5" +
    " for update " );
```

- Obtenção do nome do cursor e execução do comando de atualização.

A sintaxe SQL é da forma:

```
update <nome_da_tabela>
set <lista_de_colunas>=<lista_de_valores>
where current of <nome_do_cursor>;
```

```
String cursName = rs.getCursorName();
System.out.println( "cursor name is " + cursName );
Statement stmt2 = con.createStatement();
// update stmt2 at coll = 5
int result = stmt2.executeUpdate(
    "update loadtest set col2 = '1000' " +
    "where current of " + cursName );
```

- Revisão dos resultados

```
// recuperar linha para ver o valor atualizado
rs = stmt1.executeQuery( "select * from loadtest " +
    " where coll = 5 " );
System.out.println( "coll = " + rs.getInt( 1 ) +
    " col2 = " + rs.getInt( 2 ) );
```

A listagem completa é exibida a seguir.

```
import java.sql.*;
import java.io.*;
import java.util.Date;

class PosUpd {

    public static void main( String argv[] ) {

        try {

            Class.forName ( "jdbc.odbc.JdbcOdbcDriver" );

            String url = "jdbc:odbc:informix5";

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

            System.out.println( "Database opened." );
            Statement stmt1 = con.createStatement();

            // need a database that supports positioned updates
            DatabaseMetaData dmd = con.getMetaData();
            if ( dmd.supportsPositionedUpdate() == false ) {
                System.out.println( "Positioned update is not
                    supported by this database." );
                System.exit( -1 );
            }
        }
    }
}
```

```

ResultSet rs = stmt1.executeQuery( "select " +
    " * from loadtest where coll = 5" +
    " for update " );
rs.next(); // look at the first row (coll=5)
String cursName = rs.getCursorName();

System.out.println( "cursor name is " + cursName );

Statement stmt2 = con.createStatement();

// update stmt2 at coll = 5
int result = stmt2.executeUpdate(
    "update loadtest set col2 = '1000' " +
    " where current of " + cursName );

rs = stmt1.executeQuery( "select * from loadtest " +
    " where coll = 5 " );

rs.next();
System.out.println( "coll = " + rs.getInt( 1 ) +
    " col2 = " + rs.getInt( 2 ) );

}

catch (java.lang.Exception ex) {

// Print description of the exception.
System.out.println( "** Error on data select. ** " );
ex.printStackTrace ();

}

}

}

```

Modos de Transação

Exemplo de utilização do conceito de transações em SQL : Transdata

Introdução:

Uma transação é uma série de atualizações em um BD que são agrupadas formando uma transação única, atômica. Caso uma das atualizações falhe, as demais serão desfeitas por efeito de um **rollback**, que consiste em deixar o ambiente no mesmo estado que se encontrava quando se iniciou a transação. As transações, em JDBC, são iniciadas e terminadas por um **commit**.

Para iniciar uma transação e, em caso de falha, recuperar o estado inicial do BD, o que se faz é:

- Invocar o método **commit** de **Connection** para comprometer as atualizações que serão iniciadas.
- Escrever a série de comandos SQL que compõem a transação.
- Invocar novamente o método **commit** para comprometer a transação com o BD.
- Caso a transação falhe devido a qualquer erro, o bloco de código **catch** contém uma chamada ao método **rollback** que faz o BD retornar ao estado corrente.

A propriedade “default” do atributo “**auto commit**”, para JDBC, é verdadeira. Isto significa que cada comando SQL é processado individualmente, como uma transação unitária. Para poder utilizar transações não

unitárias é preciso desabilitar esse atributo, fazendo

con. SetAutoCommit(false);

Através do uso de modos de transação pode-se habilitar vários graus de integridade de transações para emprego durante a execução de um programa. Em um dos modos os registros não comprometidos podem ser lidos por outros usuários e os registros atualizados por uma transação também podem ser lidos por outros usuários. Em outro modo os registros comprometidos podem ser lidos por apenas uma aplicação e nenhum registro que tenha sido atualizado por uma aplicação pode ser lido por outros usuários. A granularidade aplicada às transações melhora o rendimento e permite o acesso concorrente sempre que uma aplicação o permita (como é o caso da geração de relatórios). Quando uma aplicação necessita atualizar diversas tabelas dentro de uma mesma transação ela precisa comprometer as linhas que estão sendo atualizadas limitando o acesso concorrente.

O exemplo que a ser exibido inicia com a criação de uma tabela e um índice (usando com indexador a sua primeira coluna) para essa tabela. Estes comandos SQL ocorrem antes do início da transação. Define-se, então um **PreparedStatement** com um parâmetro. Inicia-se a transação pela invocação do método **commit** (da forma **con.commit();**) e efetua-se uma repetição de inclusão de vinte linhas na tabela. Ao invés de se encerrar a transação com outra invocação do método **commit** o que se faz é invocar o método **rollback** (da forma **con.rollback();**). Este último método desfaz todas as atualizações efetuadas depois da execução do último **commit**. Como a transação só englobava a inclusão de registros e não a criação da tabela e do índice, estes objetos devem permanecer presentes embora vazios pela anulação das inclusões. A constatação desse fato é feita pela exibição do conteúdo (vazio) da tabela. O resultado da consulta não é inválido porque a tabela existe mas a resposta à consulta é vazia.

Enunciado:

Criar uma tabela **transtest** do tipo

Create table transtest (col1 int, col2 int, col3 char(10))

Os dados devem ser inseridos na tabela por um PreparedStatement e devem ser do tipo

1, 1, 'XXXXXXXX'

2, 1, 'XXXXXXXX'

3, 1, 'XXXXXXXX'

20, 1, 'XXXXXXXX'

A seguir deve-se exibir o conteúdo da tabela criada

A url é **jdbc:oracle:thin:@sbd:1521:orcl**. A conexão deve ser obtida com **getConnection url,"aluno01","aluno01"**. Antes da execução do programa deve-se entrar na conta **"aluno01","aluno01"** do Oracle e eliminar a tabela **transtest** pelo comando **drop table transtest**.

- Carga do "Driver" e criação da conexão

```
Class.forName ("oracle.jdbc.driver.OracleDriver");  
// @nome_do_computador:porta:nome_da_instância  
Connection con = DriverManager.getConnection  
    ("jdbc:oracle:thin:@sbd:1521:orcl", "aluno01", "aluno01");
```

- Ajuste do modo de transição

```
    // desligamento do modo default que é autocommit  
    // assim fazendo pode-se agrupar comandos em transações  
con.setAutoCommit( false );
```

- Verificação de existência da tabela com eventual criação

```
DatabaseMetaData dmd = con.getMetaData();  
ResultSet tablesRS = dmd.getTables(null,null,"transtest", null);  
more = tablesRS.next();
```

```

if ( more )
    System.out.println( "Tabela transtest já existe e não será criada." );
else
    { // Criação da tabela

```

- Criação de comando e execução de DDL e DML no ramo **else** do **if**

```

Statement stmt = con.createStatement();
int result = stmt.executeUpdate(
    "create table transtest( col1 int, col2 int, col3 char(10) )" );
result = stmt.executeUpdate(
    "create index idx1 on transtest( col1 ) " );

```

- Compromisso do trabalho

Se algum comando SQL provocar algum erro é lançada uma exceção **SQL Exception** e tratada pelo bloco de código **catch** do método. Este bloco executa um **rollback**. Caso a execução do código chegue à linha subsequente é porque não houve exceções fatais e os dados podem ser comprometidos no BD, na forma:

```

con.commit();

```

- Criação de Prepared Statement e execução das atualizações

```

//Inclusão de 20 registros na tabela
int n = 0;
PreparedStatement prepStmt = con.prepareStatement(
    " insert into transtest values ( ?, 1, 'XXXXXXXX' ) " );
for ( n = 1; n < 20; n++ ) {
    prepStmt.setInt( 1, n );
    prepStmt.executeUpdate();
}

```

- Exibição dos resultados da inclusão

```

Statement stmt3 = con.createStatement();
ResultSet rs1 = stmt3.executeQuery("select * from transtest");
more = rs1.next();
while ( more ) {
    System.out.println( "col1: " + rs1.getString(1) +
        " col2: " + rs1.getString(2) );
    more = rs1.next();
}

```

- Rollback e exibição dos resultados

```

System.out.println( "Restaurando dados." );
con.rollback();

Statement stmt1 = con.createStatement();
ResultSet rs = stmt1.executeQuery("select * from transtest");
more = rs.next();
if ( more == false ) // ResultSet vazio
    System.out.println( "Transação recuperada " );

```

- Bloco de código **catch**

```

catch ( java.lang.Exception ex ) {
    // Tratamento de erro.
    System.out.println( "** Erro na inclusão de dados ** " );
    ex.printStackTrace ();
}

```

A listagem completa é exibida a seguir.

```
import java.sql.*;
import java.io.*;

class TransData {

    public static void main( String argv[] ) {

        try {

            boolean more = false;

            // Carga do "driver"
            Class.forName ("oracle.jdbc.driver.OracleDriver");

            // @nome_do_computador:porta:nome_da_instância
            Connection con = DriverManager.getConnection
                ("jdbc:oracle:thin:@sbd:1521:orcl", "aluno01", "aluno01");

            // desligamento do modo default que é autocommit
            // assim fazendo pode-se agrupar comandos em transações

            con.setAutoCommit( false );

            // Determina se existe ou não a tabela
            DatabaseMetaData dmd = con.getMetaData();
            ResultSet tablesRS = dmd.getTables(null,null,"transtest",
null);
            more = tablesRS.next();

            if ( more )
                System.out.println( "Tabela transtest já existe e não será
criada." );
            else
                { // Criação da tabela e de um índice em sua primeira coluna
                    Statement stmt = con.createStatement();
                    int result =
                        stmt.executeUpdate(
"create table transtest( coll1 int, coll2 int, coll3 char(10) )" );

                    result =
                        stmt.executeUpdate("create index idx1 on transtest( coll1 )" );
                }

            con.commit();
            //Inclusão de 20 registros na tabela
            int n = 0;

            PreparedStatement prepStmt = con.prepareStatement(
                " insert into transtest values ( ?, 1, 'XXXXXXX' )" );

            for ( n = 1; n < 20; n++ ) {
                prepStmt.setInt( 1, n );
                prepStmt.executeUpdate();
            }
            //Exibição dos resultados da inclusão
            Statement stmt3 = con.createStatement();
```

```

ResultSet rs1 = stmt3.executeQuery("select * from transtest");
more = rs1.next();
while ( more ) {
    System.out.println( "col1: " + rs1.getString(1) +
        " col2: " + rs1.getString(2) );
    more = rs1.next();
}

System.out.println( "Restaurando dados." );
con.rollback();

Statement stmt1 = con.createStatement();
ResultSet rs = stmt1.executeQuery("select * from transtest");
more = rs.next();
if ( more == false ) // ResultSet vazio
    System.out.println( "Transação recuperada " );
}

catch ( java.lang.Exception ex ) {

// Tratamento de erro.
System.out.println( "*** Erro na inclusão de dados ** " );
ex.printStackTrace ();

}

}

}

```

Aplicação CGI

Exemplo de aplicações com CGI : cgiApp

Introdução:

As aplicações de CGI na Web são omnipresentes. Embora o uso de JDBC em applets pode eliminar a necessidade de muitos programas CGI mas restrições de segurança e exigências de velocidade podem tornar CGI alternativas válidas. Java, sendo uma linguagem flexível serve para criar estas CGI.

Se for o caso da applet ou página HTML ser conectada a um BD em outro servidor que não o servidor Web, existem razões para não expor o servidor de BD à Web e dar preferência a que o servidor HTTP processe e gerencie a conexão.

Para conectar-se ao servidor de BD uma aplicação de três camadas é necessária. Uma aplicação CGI é uma abordagem interessante para esta terceira camada. Esta aplicação CGI pode receber uma requisição, recuperar os dados e formatá-los para retorno à página HTML.

Enunciado:

Construir uma aplicação para recuperação dos registros de uma tabela de usuários na qual o sobrenome é da forma de um parâmetro passado para o programa (CGI). O programa recebe os argumentos da linha de comando ou "CGI token". Para a recuperação usa-se um objeto **StringTokenizer** e um objeto **Vector**. O objeto **StringTokenizer Params** recebe como parâmetros a lista de argumentos da linha de comando, **argv[0]** e um caractere delimitador (por exemplo "+") que separa, no string de entrada, um parâmetro de outro. Faz-se uma repetição e os parâmetros são adicionados a um objeto **Vector** que os armazenará e depois os passará para a consulta. O "token" é analisado e usado como parâmetro em um comando SQL. O resultado do comando SQL é formatado como página HTML e exibido na tela.

O HTML pode ser do tipo

```
</ul>;
```

```
<p> Customer address information is listed in the table below </p>;
```

```
// Table header
<table border > ;
<caption>Customer Addresses </caption> ;
<th> First Name </th>;
<th> Last Name </th>;
<th> Address </th> ;
<th> City </th> ;
<th> State </th> ;
<th> Zip </th> ;
```

O comando SQL deve ser do tipo

Select * from funcionario where nome like ?

A url é **jdbc:odbc:msaccessdb**. A conexão deve ser obtida com **getConnection url,"Carla","Dalboni"**

- Carga do “driver” e criação da conexão

```
Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );
String url = "jdbc:odbc:msaccessdb";
// Cria uma conexao utilizando como parametro a URL, Login e Senha.
Connection con = DriverManager.getConnection (url, "Carla", "Dalboni");
```

- Criação de um **PreparedStatement** com parâmetros

```
PreparedStatement stmt = con.prepareStatement(
    " select * from funcionario " +
    " where nome like ? " );
```

- Análise dos argumentos CGI

Os argumentos CGI são passados ao programa usando um “+” para separar os argumentos.

```
// Concatena argumentos CGI
if ( argv.length == 0 ) {
    System.out.println( "Parametros Invalidos. Programa abortado. " );
    System.exit( -1 );
}
```

```
StringTokenizer Params = new StringTokenizer( argv[0], delim );
Vector vParams = new Vector();
String s = null;
```

```
while ( Params.hasMoreTokens() ) {
    s = Params.nextToken();
    vParams.addElement( s );
}
```

- Ajuste de parâmetros e execução da consulta

```
// Arg1 é o ultimo nome
s = vParams.elementAt(0).toString();
stmt.setString( 1, vParams.elementAt( 0 ).toString() );
// Cria um ResultSet a partir da clausula SQL que foi preparada
ResultSet rs = stmt.executeQuery();
System.out.println("Adicionado: " + vParams.elementAt( 0 ).toString());
```

- Recuperação dos resultados e exibição da saída formatada

```
ResultSetMetaData rsmd = rs.getMetaData();

boolean more = rs.next();
if ( !more ) {
    System.out.println( "Error - nenhum dado retornado" );
    System.exit( -1 );
}
```

```

// cabeçalho de pagina html
System.out.println( "</ul>" );
System.out.println( "<p> Customer address information is listed in the
table below </p>" );

// cabeçalho de tabela - html
System.out.println( "<table border > " );
System.out.println( "<caption>Customer Addresses </caption> " );
System.out.println( "<th> First Name </th>" );
System.out.println( "<th> Last Name </th>" );
System.out.println( "<th> Address </th> " );
System.out.println( "<th> City </th> " );
System.out.println( "<th> State </th> " );
System.out.println( "<th> Zip </th> " );

// Loop para preencher conteudo de tabela.
while ( more ) {
    System.out.println( "<tr> " );
    for ( n = 1; n <= rsmd.getColumnCount(); n++ )
        System.out.println( "<td > " +
            rs.getString( n ) +
            " </td> " );
    System.out.println( "</tr>" );
    more = rs.next();
}
// Finaliza comando de tabela.
System.out.println( "</table> " );
}

```

A listagem completa é exibida a seguir.

```

import java.sql.*;
import java.util.StringTokenizer;
import java.util.Vector;
import java.io.*;

class cgiApp {

    static String delim = "+";

public static void main( String argv[] ) {

    int n = 0;

    try {

        Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );

        String url = "jdbc:odbc:msaccessdb";
        // Cria uma conexao utilizando como parametro a URL, Login e Senha.
        Connection con = DriverManager.getConnection (
            url, "Carla", "Dalboni" );

        // Prepara a execucao de um comando SQL.
        PreparedStatement stmt = con.prepareStatement(
            " select * from funcionario " +
            " where nome like ? " );
    }
}

```

```

// Concatena argumentos CGI

if ( argv.length == 0 ) {
    System.out.println( "Parametros Invalidos. Programa abortado. " );
    System.exit( -1 );
}

StringTokenizer Params = new StringTokenizer( argv[0], delim );

Vector vParams = new Vector();
String s = null;

while ( Params.hasMoreTokens() ) {

    s = Params.nextToken();
    vParams.addElement( s );
}

// Arg1 é o ultimo nome
s = vParams.elementAt(0).toString();

stmt.setString( 1, vParams.elementAt( 0 ).toString() );

// Cria um ResultSet a partir da clausula SQL que foi preparada.
ResultSet rs = stmt.executeQuery();

System.out.println("Adicionado: " + vParams.elementAt( 0
).toString());

ResultSetMetaData rsmd = rs.getMetaData();

boolean more = rs.next();
if ( !more ) {
    System.out.println( "Error - nenhum dado retornado" );
    System.exit( -1 );
}

// cabecalho de pagina html
System.out.println( "</ul>" );
System.out.println( "<p> Customer address information is listed in
the table below </p>" );

// cabecalho de tabela - html
System.out.println( "<table border > " );
System.out.println( "<caption>Customer Addresses </caption> " );
System.out.println( "<th> First Name </th>" );
System.out.println( "<th> Last Name </th>" );
System.out.println( "<th> Address </th> " );
System.out.println( "<th> City </th> " );
System.out.println( "<th> State </th> " );
System.out.println( "<th> Zip </th> " );

// Loop para preencher conteudo de tabela.
while ( more ) {
    System.out.println( "<tr> " );

```

```
        for ( n = 1; n <= rsmd.getColumnCount(); n++ )
            System.out.println( "<td > " +
                                rs.getString( n ) +
                                " </td> " );
        System.out.println( "</tr>" );
        more = rs.next();
    }
    // Finaliza comando de tabela.
    System.out.println( "</table> " );
}

catch ( java.lang.Exception ex ) {
    ex.printStackTrace();
}

}

}
```

Acesso a Metadados

Exemplo de utilização de meta dados : `MetaDataExample1`

Introdução:

Os meta dados são dados a respeito do conteúdo de um **ResultSet** retornado. Através dos meta dados pode-se conhecer a estrutura da tabela original e/ou do resultado da consulta. Como exemplo de exploração dos meta dados será apresentado um programa de demonstração. Este programa é semelhante aos exercícios iniciais. A diferença é a exploração, em maior escala, da transformação dos tipos de dados SQL em classes Java e sua exibição, pois nos exercícios iniciais tratavam-se todos os campos como strings.

O procedimento **formatOutputString** é usado para interpretar e fornecer os dados. Este procedimento recebe três parâmetros (o objeto **ResultSetMetaData**, o objeto **ResultSet** e o índice da coluna da qual se deseja a saída formatada). **OutputString** é o string retornado pelo método e **colTypeNameString** é o string usado para armazenar o nome do tipo de dado do tipo da coluna. O método **getColumnType** de **ResultSetMetaData** é usado para recuperar o tipo de coluna. Este método chama o método **formattedValue** para formatar os dados na coluna baseado no tipo de dado da coluna. Se o objeto retornado não for nulo um string é criado com

- nome da coluna
- nome do tipo de dados
- valor do objeto na forma retornada pelo método **toString**.

O método **typeNameString** toma um inteiro retornado pelo método **getColumnType** e mapeia esse inteiro em um string. O método **formattedValue** retorna uma referência a um objeto. Obtido o tipo de dado do objeto, o método **get** adequado do **ResultSet** é chamado para recuperar o dado. O dado vem com o tipo identificado mas é moldado (“cast”, da forma **(Object) booleanObj** ou **(Object) integerObj**) como uma referência a objeto.

Enunciado:

Exibir o conteúdo de uma tabela. O programa deve receber uma consulta, entre aspas duplas, como argumento de linha de comando. A exibição de registros deve ser com uma linha por atributo de cada registro, da forma

Col <número da coluna> **O tipo de dado** é <tipo de dado da coluna> **Valor** : <Valor do atributo>

Deve-se converter os tipos de dados SQL para objetos Java antes da formatação de saída.

A url é **jdbc:odbc:msaccessdb**. A conexão deve ser obtida com **getConnection url,"Carla","Dalboni"**

A interface JDBC dá acesso a uma ampla gama de informações sobre o BD corrente ou **ResultSet**. A classe **ResultSetMetadata** traz, entre outras, informações sobre o número e tipos de colunas retornadas.

- Recuperação da consulta da linha de comando

```
// A consulta default é NULL
String queryString = null;
```

```
// O data source name default é
String url = "jdbc:odbc:msaccessdb";
```

```
// O primeiro argumento da linha de comando é a consulta
if ( argv.length > 0 )
    queryString = argv[0];
```

```
// Se não houver consulta o programa deve terminar
if ( queryString == null ) {
    System.out.println
        ( "Programa abortando. Deveria ter recebido uma consulta. " );
    System.exit(-1);
}
```

- Carga do "driver" e criação da conexão

```
Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );
Connection con = DriverManager.getConnection ( url, "Carla", "Dalboni" );
```

- Criação do comando e execução da consulta

```
Statement stmt = con.createStatement( );
ResultSet rs = stmt.executeQuery( queryString );
```

- Recuperação do ResultSet e determinação do número de colunas

```
// Determinando a natureza dos resultados
ResultSetMetadata md = rs.getMetaData();
// Exibindo os resultados
int numCols = md.getColumnCount();
System.out.println("");
```

- Execução da rotina de formatação

- // Rotina de formatação

```
static String formatOutputString
    ( ResultSetMetadata rsmd, ResultSet rs, int colIndex ) {
    String outputString = null;
    String colTypeNameString = null;

    try {

        int colType = rsmd.getColumnType( colIndex );

        colTypeNameString = typeNameString( colType );
```

```

if ( colTypeNameString.equals( "UNKNOWN" ) ||
    colTypeNameString.equals( "OTHER" ) )
    colTypeNameString = rsmd.getColumnTypeName( colIndex );

Object obj = formattedValue( rs, rsmd, colIndex, colType );
if ( obj == null )
    return ( " ** NULL ** " );

OutputString = "Coluna: " + rsmd.getColumnLabel( colIndex ) +
    " O tipo de dado eh: " + colTypeNameString +
    " ; o valor eh: " + obj.toString();

// Obtenção do tipo real de dado e retorno como objeto não como string

    / rs.getString( colIndex );
    }

catch ( SQLException ex ) {
    System.out.println ( "\n*** Capturada SQLException ***\n");
    while ( ex != null ) {
        System.out.println ( "SQLState: " + ex.getSQLState ());
        System.out.println ( "Message: " + ex.getMessage ());
        System.out.println ( "Vendor: " + ex.getErrorCode ());
        ex = ex.getNextException ();
        System.out.println ( "");
    }
}

return( OutputString );

}

// Retornar nome do tipo como string

static String typeNameString( int Type ) {

    switch ( Type ) {

        case ( Types.BIGINT ):           return ( "BIGINT" );
        case ( Types.BINARY ):           return ( "BINARY" );
        case ( Types.BIT ):              return ( "BIT" );
        case ( Types.CHAR ):             return ( "CHAR" );
        case ( Types.INTEGER ):          return ( "INTEGER" );
        case ( Types.DATE ):             return ( "DATE" );
        case ( Types.DECIMAL ):          return ( "DECIMAL" );
        case ( Types.FLOAT ):            return ( "FLOAT" );
        case ( Types.LONGVARBINARY ) :   return ( "LONGVARBINARY" );
        case ( Types.LONGVARCHAR ) :     return ( "LONGVARCHAR" );
        case ( Types.OTHER ) :           return ( "OTHER" );

    }

    return ( "UNKNOWN" );

}

static Object formattedValue( ResultSet rs,
                             ResultSetMetaData rsmd,

```

```

        int colIndex,
        int Type ) {

Object generalObj = null;

    try {

switch ( Type ) {

case ( Types.BIGINT ):
    Long longObj = new Long( rs.getLong( colIndex ) );
    return ( Object ) longObj );
case ( Types.BIT ):
    Boolean booleanObj = new Boolean( rs.getBoolean( colIndex ) );
    return ( Object ) booleanObj );
case ( Types.CHAR ):
    String stringObj = new String( rs.getString( colIndex ) );
    return ( Object ) stringObj );
case ( Types.INTEGER ):
    Integer integerObj = new Integer( rs.getInt( colIndex ) );
    return ( Object ) integerObj );
case ( Types.DATE ):
    Date dateObj = rs.getDate( colIndex );
    return ( Object ) dateObj );
case ( Types.DECIMAL ):
case ( Types.FLOAT ):
    System.out.println(rs.getBigDecimal
        ( colIndex, rsmd.getScale( colIndex ) ));
    BigDecimal numericObj = rs.getBigDecimal( colIndex,
        rsmd.getScale( colIndex ) );
    return ( Object ) numericObj );

case ( Types.BINARY ):
case ( Types.LONGVARBINARY ) :
case ( Types.LONGVARCHAR ) :
case ( Types.OTHER ) :
    return ( rs.getObject( colIndex ) );

}
// Obter o manipulador de objeto
generalObj = rs.getObject( colIndex );

}

catch ( SQLException ex ) {

    System.out.println ( "\n*** SQLException caught ***\n" );

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

}

// Retorna referencia ao objeto
return ( generalObj );

```

```
}
```

- Iteração sobre os resultados exibindo dados formatados

```
•  
// Exibição dos dados até o final do ResultSet  
    boolean more = rs.next();  
    int rowCount = 0;  
    while (more) {  
  
        rowCount++;  
        System.out.println( "*** linha " + rowCount + " *** " );  
            // Repetição em cada coluna, obtendo  
            // e exibindo os dados da coluna  
        for (n=1; n<=numCols; n++) {  
            // ** chamada da rotinas de formatação **  
            System.out.println( formatOutputString( md,rs, n ) );  
  
        }  
        System.out.println("");  
        more = rs.next();  
    }  
}
```

A listagem completa é exibida a seguir.

```
import java.net.URL;  
import java.sql.*;  
import java.math.*;  
import java.io.*;
```

```
class MetaDataExample1 {  
  
    public static void main( String argv[] ) {  
  
        short n = 0;  
  
        try {  
  
            // A consulta default é NULL  
            String queryString = null;  
  
            // O data source name default é  
            String url    = "jdbc:odbc:msaccessdb";  
  
            // O primeiro argumento da linha de comando é a consulta  
            if ( argv.length > 0 )  
                queryString = argv[0];  
  
            // Se não houver consulta o programa deve terminar  
            if ( queryString == null ) {  
                System.out.println( "Programa abortando. Deveria ter recebido uma  
consulta. " );  
                System.exit(-1);  
            }  
  
            // Carregando o "driver" e criando um conexão  
            Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

```

Connection con = DriverManager.getConnection (
    url, "Carla", "Dalboni");

    // Criando um comando
Statement stmt = con.createStatement( );

    // Executando a consulta
ResultSet rs = stmt.executeQuery( queryString );

    // Determinando a natureza dos resultados
ResultSetMetaData md = rs.getMetaData();

    // Exibindo os resultados

int numCols = md.getColumnCount();

System.out.println("");

    // Exibição dos dados até o final do ResultSet
boolean more = rs.next();
int rowCount = 0;
while (more) {

    rowCount++;
    System.out.println( "*** linha " + rowCount + " *** " );
        // Repetição em cada coluna, obtendo
        // e exibindo os dados da coluna
    for (n=1; n<=numCols; n++) {
        // ** chamada da rotinas de formatação **
        System.out.println( formatOutputString( md,rs, n ) );
    }
    System.out.println("");
    more = rs.next();
}
}
catch ( SQLException ex ) {
    System.out.println ( "\n*** Capturada SQLException ***\n");
    while (ex != null) {
        System.out.println ( "SQLState: " + ex.getSQLState ());
        System.out.println ( "Mensagem: " + ex.getMessage ());
        System.out.println ( "Vendedor: " + ex.getErrorCode ());
        ex = ex.getNextException ( );
        System.out.println ( "");
    }
}
catch ( java.lang.Exception ex) {

    // Capturada outra exceção. Ela será exibida.

    ex.printStackTrace ( );
}
}

// Rotina de formatação
static String formatOutputString

```

```

    ( ResultSetMetaData rsmd, ResultSet rs, int colIndex ) {
String OutputString = null;
String colTypeNameString = null;

try {

    int colType = rsmd.getColumnType( colIndex );

    colTypeNameString = typeNameString( colType );
    if ( colTypeNameString.equals( "UNKNOWN" ) ||
        colTypeNameString.equals( "OTHER" ) )
        colTypeNameString = rsmd.getColumnTypeName( colIndex );

    Object obj = formattedValue( rs, rsmd, colIndex, colType );
    if ( obj == null )
        return ( " ** NULL ** " );

    OutputString = "Coluna: " + rsmd.getColumnLabel( colIndex ) +
        " O tipo de dado eh: " + colTypeNameString +
        " ; o valor eh: " + obj.toString();

// Obtenção do tipo real de dado e retorno como objeto não como string

/ rs.getString( colIndex );
}

catch ( SQLException ex ) {
    System.out.println ( "\n*** Capturada SQLException ***\n");
    while ( ex != null ) {
        System.out.println ( "SQLState: " + ex.getSQLState ());
        System.out.println ( "Message: " + ex.getMessage ());
        System.out.println ( "Vendor: " + ex.getErrorCode ());
        ex = ex.getNextException ();
        System.out.println ( "" );
    }
}

return( OutputString );

}

// Retornar nome do tipo como string

static String typeNameString( int Type ) {

    switch ( Type ) {

        case ( Types.BIGINT ) :           return ( "BIGINT" );
        case ( Types.BINARY ) :           return ( "BINARY" );
        case ( Types.BIT ) :              return ( "BIT" );
        case ( Types.CHAR ) :             return ( "CHAR" );
        case ( Types.INTEGER ) :          return ( "INTEGER" );
        case ( Types.DATE ) :             return ( "DATE" );
        case ( Types.DECIMAL ) :          return ( "DECIMAL" );
        case ( Types.FLOAT ) :            return ( "FLOAT" );
        case ( Types.LONGVARBINARY ) :    return ( "LONGVARBINARY" );
        case ( Types.LONGVARCHAR ) :      return ( "LONGVARCHAR" );
        case ( Types.OTHER ) :            return ( "OTHER" );
    }
}

```

```

    }

    return ( "UNKNOWN" );
}

static Object formattedValue( ResultSet rs,
                             ResultSetMetaData rsmd,
                             int colIndex,
                             int Type ) {

    Object generalObj = null;

    try {

        switch ( Type ) {

            case ( Types.BIGINT ):
                Long longObj = new Long( rs.getLong( colIndex ) );
                return ( Object ) longObj ;
            case ( Types.BIT ):
                Boolean booleanObj = new Boolean( rs.getBoolean( colIndex ) );
                return ( Object ) booleanObj ;
            case ( Types.CHAR ):
                String stringObj = new String( rs.getString( colIndex ) );
                return ( Object ) stringObj ;
            case ( Types.INTEGER ):
                Integer integerObj = new Integer( rs.getInt( colIndex ) );
                return ( Object ) integerObj ;
            case ( Types.DATE ):
                Date dateObj = rs.getDate( colIndex );
                return ( Object ) dateObj ;
            case ( Types.DECIMAL ):
            case ( Types.FLOAT ):
                System.out.println(rs.getBigDecimal
                    ( colIndex, rsmd.getScale( colIndex ) ));
                BigDecimal numericObj = rs.getBigDecimal( colIndex,
                    rsmd.getScale( colIndex ) );
                return ( Object ) numericObj ;

            case ( Types.BINARY ):
            case ( Types.LONGVARBINARY ) :
            case ( Types.LONGVARCHAR ) :
            case ( Types.OTHER ) :
                return ( rs.getObject( colIndex ) );

        }

        // Obter o manipulador de objeto
        generalObj = rs.getObject( colIndex );

    }

    catch ( SQLException ex ) {

        System.out.println ( "\n*** SQLException caught
***\n" );

        while ( ex != null ) {

```

```

        System.out.println ("SQLState: " + ex.getSQLState ());
        System.out.println ("Message: " + ex.getMessage ());
        System.out.println ("Vendor: " + ex.getErrorCode ());
        ex = ex.getNextException ();
        System.out.println ("");
    }
}

// Retorna referencia ao objeto
return ( generalObj );
}
}

```

Percurso sobre o “array” ResultSet

Um objeto da Classe **RSArray** é declarado para conter os elementos do **ResultSet** retornados pelo objeto **Statement**. Esta classe **RSArray** contém métodos para tratar de objetos passados por referência. Estes objetos são armazenados em dois objetos **Vector**. Um objeto **Vector** é usado para armazenar o objeto ponteiro **ResultSet** e outro, o **ResultsBufferVector**, que contém as colunas. A declaração é da forma:

```
static RSArrayGen rsBuff = new RSArrayGen();
```

A definição da classe **RSArray** pode ser vista a seguir:

```

class RSArray1 {
    // variáveis de instâncias
    int index = 0;
    // vetor do conjunto de resultados
    Vector ResultsBuffer = new Vector();
    // vetor de linhas
    Vector columns = new Vector();
.
.
    // agora vem os métodos da classe
    void addElement (ResultSet rs) {
        int x;
        try {
            // armazenamento das colunas em um vetor
            for (x = 1; x <= rs.getMetaData.getColumnCount(); x++)
                columns.addElement ((Object)rs.getObject(x));
            // armazenamento do vetor de colunas no vetor Results
            ResultsBuffer.addElement ((Object) columns.clone());
            columns.removeAllElements();
        }
        catch(java.lang.Exception ex) {
            ex.printStackTrace ();
        }
    }
}

```

```

Object ElementAt (int target Index)    {
// retorna o objeto que está na posição Index do ResultBuffer
Vector returnVector = null;
try    {
    returnVector=(Vector)ResultsBuffer.elementAt(targetIndex x-1);
    }
catch (java.lang.Exception ex)    {
    ex.printStackTrace();
    }
return ((Object) returnVector);
}
Object next()    {
// retorna o próximo elemento sequencial de RSArray
    index ++;
    return (ElementAt(index));
}
Object previous()    {
// retorna o elemento anterior de RSArray
    index --;
    return (ElementAt (index));
}
}

```

Exemplo de Percurso sobre o “array” ResultSet

Para criar a **classe RSArray** o que se faz é o seguinte:

```

import java.sql.*;
import java.io.*;
import java.util.Date;
import java.util.Vector;

class RSArrayGen {
    // variáveis de instância
    int index = 0;

    // vetor def ResultSets
    Vector ResultsBuffer = new Vector();

    // Vetor de linha contém uma tupla, armazenando as colunas dela
    Vector columns = new Vector();

    // ---Inclusão de elementos no Array-----
    void addElement( ResultSet rs ) {
        int x;

        try {

            // armazenas as colunas em um vetor
            for ( x = 1;
                x <= rs.getMetaData().getColumnCount();
                x++ )
                columns.addElement( (Object) rs.getObject( x ) );
        }
    }
}

```

```

        // armazena o vetor de cdolunas no vetor de ResultSets
        ResultsBuffer.addElement( (Object) columns.clone() );
        columns.removeAllElements();

    }

    catch ( java.lang.Exception ex ) {

        ex.printStackTrace();

    }
}

// ---- Acesso a elementos do Array
Object ElementAt( int targetIndex ) {

    Vector returnVector = null;

    try {

        returnVector = (Vector) ResultsBuffer.elementAt( targetIndex-1 );
    }
    catch ( java.lang.Exception ex ) {
        ex.printStackTrace();
    }
    return ( (Object) returnVector );
}

// ---- Acesso ao próximo elemento do Array
Object next() {
    index++;
    return ( ElementAt( index ) );
}

// ---- Acesso ao elemento anterior do Array
Object previous() {
    index--;
    return ( ElementAt( index ) );
}
}

```

Para poder ter liberdade de movimento sobre o “array” **ResultSet** os passos a adotar são os seguintes, quando se quiser explorar a **classe RSArray**.

- Declaração do objeto **RSArray**

```
static RSArray rsBuff = new RSArray();
```
- Carga do **DriverManager** e conexão

```
Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );
String url = "jdbc:odbc:msaccessdb";
Connection con = DriverManager.getConnection (url, " ", " ");
```
- Criação e execução de comando

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery ("select * from funcionario");
```

- Iteração sobre o **ResultSet** adicionando ao Buffer **ResultSetArray**

```
while (rs.next && n ++ < 50)  {  
  // carregar 50 registros  
  rsBuff.addElement (rs);  
}  
int rowsLoaded = n;
```

- Exibição dos resultados

```

System.out.println ("Processadas" + n + "linhas");
// percurso sobre o vetor rs buffer ResultsBuffer
Vector columnsVector = null;
for (x = 0; x < rowsLoaded - 1; x ++) {
    // obter a linha
    columnsVector = (Vector) rsBuff.ElementAt (x + 1);
    // exibição do conteúdo das linhas
    for (n = 0; n < rs.getMetaData().getColumnCount(); n ++) {
        System.out.println ("Linha" + x + "Coluna:" +
            + n + " " + columnsVector.elementAt(n).toString());
    }
}

```

A listagem completa é exibida a seguir.

```

import java.sql.*;
import java.io.*;
import java.util.Vector;

class RSArrayTest {

    public static void main (String argv[]) {

        RSArray rsBuff = new RSArray();

    try {
        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        String url = "jdbc:odbc:msaccessdb";
        Connection con = DriverManager.getConnection (url, "", "");

        // Criação de comando
        Statement stmt = con.createStatement ();

        // Execução do comando
        String gs = " select * from Empregados ";
        ResultSet rs = stmt.executeQuery (gs);

        // iteração sobre os resultados
        int n = 0;
        while (rs.next() && n++ < 50) {
            rsBuff.addElement (rs);
        }

        int rowsLoaded = n;
        System.out.println ("Processadas " + rowsLoaded + " linhas.");
        System.out.println ("");
        Vector columnsVector = null;
        int x;
        for (x=0; x < rowsLoaded; x++) {
            columnsVector = (Vector) rsBuff.ElementAt(x);
            for (n=0; n < rs.getMetaData().getColumnCount(); n++) {
                System.out.println ("Linha " + x + " Coluna " + n +
                    " " + columnsVector.elementAt(n).toString());
            }
            System.out.println ("");
        }
    }
    catch (java.lang.Exception ex) {

```

```

        System.out.println ( "*** Erro na selecao dos dados ***" );
        ex.printStackTrace ();
    }
}

```

Pode-se chegar aos mesmos resultados sem criar, de maneira explícita, a classe RSArray, com um programa como o que se segue.

```

import java.sql.*;
import java.io.*;
import java.util.Date;
import java.util.Vector;

class RSArray {

public static void main( String argv[] ) {

// Criação de um vetor de result sets (tabela)
Vector ResultsBuffer = new Vector();

// Vetor de linha
// contém uma tupla, armazenando as colunas dela
Vector columns = new Vector();

    try {

        Class.forName ( "sun.jdbc.odbc.JdbcOdbcDriver" );

        String url = "jdbc:odbc:msaccessdb";
        //Cria conexao
        Connection con = DriverManager.getConnection ( url, "Carla",
"Dalboni" );

        Statement stmt = con.createStatement();

        //Executa query atraves de um resultset
        ResultSet rs    = stmt.executeQuery( " select * from funcionario" );
        int n = 0;
        int x = 1;
        ResultSetMetaData rsmd = rs.getMetaData();

        boolean more = rs.next();
        int colCount = rsmd.getColumnCount();

        while ( more && n++ < 50 ) {

            for ( x = 1; x <= colCount; x++ )
                // Cada elemento do ResultSet vai para uma posição de column
                columns.addElement( rs.getObject( x ) );
            // columns.clone() contém uma cópia de columns
            // esta cópia é adicionada ao vetor de ResultSets
            ResultsBuffer.addElement( (Vector) columns.clone() );
            columns.removeAllElements(); // Limpa o vetor columns
            more = rs.next();
        }
    }
}

```

```

int rowsLoaded = n;

System.out.println( "Processadas " + n + " linhas" );

    // Exibe o conteudo do vetor ResultsBuffer
    for ( x = 1; x < rowsLoaded; x++ )
        for ( n = 1; n <= colCount; n++ )
            System.out.println( "Linha: " + x + " Coluna: " + n + " valor: " +
( (Vector) ResultsBuffer.elementAt( x-1 ) ).elementAt( n-1 ).toString() );

    }
    catch ( java.lang.Exception ex ) {

// Tratamento de erro.
System.out.println( "*** Erro na seleção de dados *** " );
ex.printStackTrace ();
    }
}
}

```

“Applets” JDBC

Em “applets”, como na maioria das aplicações GUI, o usuário controla o fluxo do programa. O acesso ao BD é controlado pelos controles da GUI. Nessas aplicações as chamadas JDBC são usualmente feitas como resultado de disparo de eventos de botões. As chamadas JDBC são feitas nos manipuladores de botões ou em métodos chamados pelos manipuladores de botões.

Os “applets” Java usualmente tem limitações de segurança que restringem seu acesso a “sites” remotos do BD. Os “applets” Java só podem fazer conexões com o “site” de onde foram “baixados”. Se o servidor do BD desejado residir na mesma máquina em que reside o servidor que trata as páginas HTML do “applet” não existe restrição de acesso. Em caso contrário, é necessário utilizar uma arquitetura em três camadas. Neste caso o “applet” se comunica com uma aplicação em um servidor a que tenha acesso e este servidor se comunica com o servidor de BD em outra máquina.

Exemplo de “Applet JDBC

Considere-se uma aplicação navegando em uma lista de nomes exibindo os atributos **nome** e **sobrenome** na janela de “applet”. O usuário pode acionar o botão **get data** para recuperar todas as linhas do BD. O manipulador do botão constrói uma “string” de consulta que é executada e seus resultados são carregados em um objeto **Vector**. À medida que o usuário avança ou retrocede sobre os resultados usando os botões na janela, os elementos de **Vector** são recuperados e exibidos na janela. Um botão adicional serve para inclusão de dados no BD. O usuário controla o “applet” usando os botões na janela.

Botão	Características
INSERT DATA	Este botão recupera dados de campos de dados. Estes dados são colocados em um comando SQL insert e o comando é executado.
GET DATA	Esta operação recupera todos os dados na tabela do BD. Um comando SQL é criado para selecionar dados da tabela alvo. O comando SQL é executado e o resultado retorna em um ResultSet . Todo o ResultSet é armazenado em um objeto Vector . Os dados da primeira linha são exibidos nos campos da janela. Uma variável indexadora ou “ponteiro” é inicializada para indicar a posição corrente no conjunto resultado.
NEXT ROW	Esta operação incrementa a variável índice, recupera o dado na posição do objeto Vector indexada pela variável e exibe o dado na janela do “applet”.
PREVIOUS ROW	Esta operação decrementa a variável índice, recupera o dado na posição do objeto Vector indexada pela variável e exibe o dado na janela do “applet”.

Código de um “Applet”

Declarações

O controle da interação com o BD é feito pela declaração de duas classes, uma de resultados (**DBResults**) e outra de controles (**DBControl**).

A classe de controles contém os objetos para a interação com o BD. Os objetos **Connection**, **Statement** e **ResultSet**, usados pelo “applet” são variáveis de instâncias nesta classe. Os objetos são inicializados na manipuladora de eventos do botão “get data”.

São também componentes dessa classe uma variável índice para a posição corrente e um objeto **Vector**.

A classe **DBResults** contém dois Strings que correspondem a campos de dados usados para exibir os dados. Um ciclo **while** usado para ler o **ResultSet** instancia uma série de objetos do tipo **DBResults**. Um objeto é instanciado para cada linha no ciclo **while**, são inseridos dados nos objetos para cada coluna do **ResultSet** e o objeto inserido em **Vector**.

A classe **DBControl** contém declarações de objetos para controlar a conexão com o BD, o comando SQL, o **ResultSet**, uma variável índice para a posição corrente e o número máximo de linhas armazenadas. Um objeto **Vector** também é declarado para armazenar o resultado da consulta.

Exibição da janela do “Applet”

O código de exibição da janela do “applet” contém declarações para os objetos AWT usados para exibir a janela. Um método **init**, chamado quando o “applet” é disparado, inicializa a janela e exibe a tela para ser usada na navegação dos dados.

A classe **screen** contém rótulos para os campos de texto e os botões. Cada janela é declarada como classe única para o objeto janela. No construtor para a classe **Screen1** os objetos de janela são instanciados.

Declaração da Classe Button

Uma subclasse da classe **Button** é declarada com componentes para definir as propriedades visuais dos botões.

Manipulador de acionamento do botão Previous

A versão AWT Java de um manipulador de evento é usada para manipular o evento “acionamento de botão”. Sendo este botão de retrocesso, a posição corrente (**DBControl.currpos**) é decrementada e testada. Se o resultado for válido (> 1) o dado é recuperado e exibido.

Manipulador de acionamento de botão InsertData

Inicialmente prepara-se um “string” para inserção de dados na tabela do BD. Os valores são lidos dos campos de texto na janela e colocados em um comando **insert**. Com o “string” montado utiliza-se o método **executeUpdate** para inclusão dos dados no BD.

Evento de Botão NextRow

A variável índice é incrementada e testada para verificar se não ultrapassou o número máximo de linhas armazenadas no objeto **DBControl**. Se o teste indicar que o domínio foi respeitado, o valor correspondente é recuperado e exibido na janela.

Evento de Botão GetData

Começa-se estabelecendo uma conexão a BD e construindo uma consulta para recuperar todos os dados da tabela do BD. A posição inicial do índice recebe o valor 1 e a consulta é executada com o resultado armazenado em **ResultSet**. Um ciclo **while** com um método **next** percorre o **ResultSet** para recuperar os dados. Em cada iteração dados são recuperados de **ResultSet** e armazenados nos componentes **TextField** de um objeto **DBResults**. Este objeto é incluído no **Vector ResultsStorage**. O número de iterações é armazenado na instância da variável **maxrows**. O índice aponta para a primeira linha e esta será exibida na janela. A recuperação de elementos é feita pelo método **elementAT**. A exibição na janela é feita pelo método **setText**, uma vez para cada campo.

Componentes e Java Beans

Introdução

Java Beans é uma arquitetura de componentes para Java. O seu objetivo é o de habilitar os fornecedores independentes, ou “independent software vendors”, ou ISV, a desenvolver componentes reusáveis de “software” e usuários finais a combinar esses elementos usando ferramentas de construção de aplicações. O modelo de componentes Java possui dois elementos principais: **componentes** e “**containers**”. Um **componente** pode ter complexidade variando de um “widget” de GUI até uma aplicação completa. Um “**container**” agrupa componentes e pode também ser usado como um **componente** dentro de outro “**container**”. Um **componente** pode publicar ou registrar sua interface permitindo ser disparado por chamadas ou eventos de outros **componentes** ou “**scripts**”.

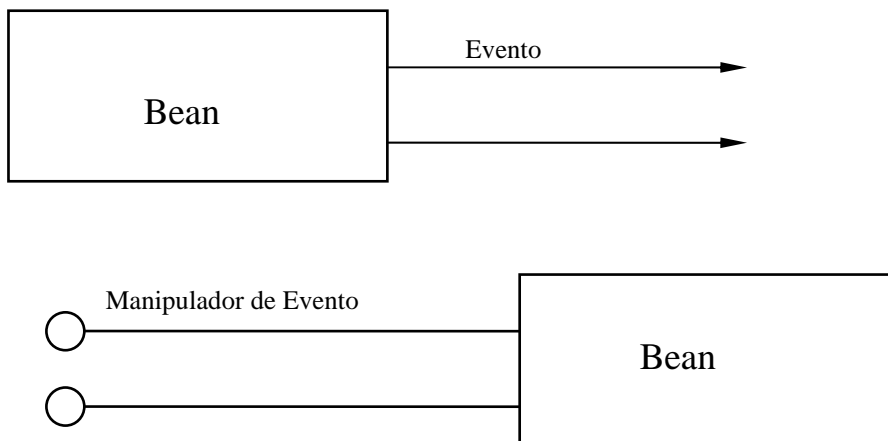
Um grão de Java, ou **Java Bean**, é um componente de “software” reutilizável, que pode ser manipulado visualmente em uma ferramenta de construção.

Uma ferramenta de construção é um ambiente de “software” no qual se pode montar **Java Beans**, dentre os quais um construtor de aplicações visuais, um construtor de páginas da Web e editores de documentos.

Um produto de desenvolvimento de **Java Beans** é o “**Beans Development Kit**”, ou **BKD**. Pode-se montar grãos de Java utilizando ferramentas de construção e utilizar grãos em navegadores sem necessidade do **BKD**.

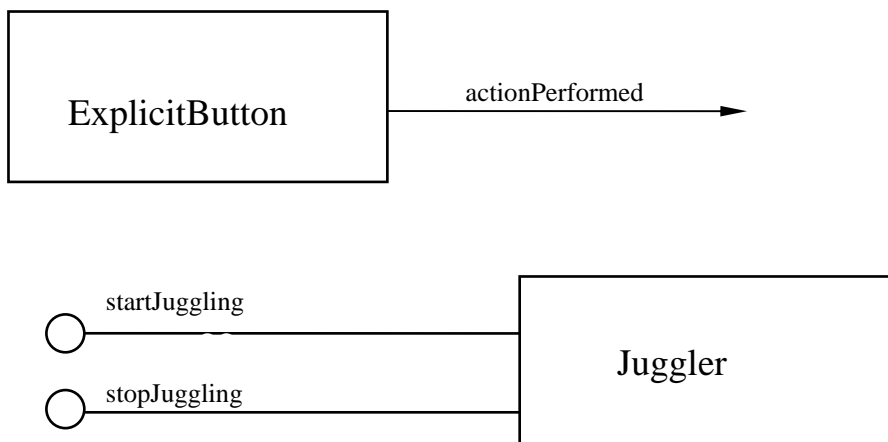
Uma das ferramentas de construção é a **BeanBox**, composta de três elementos: **BeanBox**, **ToolBox** e **PropertySheet**. A **ToolBox** contém uma lista de todos os grãos conhecidos pelo **BeanBox** (armazenado em /bdk/java de **BKD**). **BeanBox** é um “container”, em formato livre, que permite aos grãos serem alocados e manipulados. **BeanBox** é também um grão, demonstrando que grãos podem conter outros grãos. **PropertySheet** exhibe todas as propriedades editáveis para o grão corrente.

Em representação gráfica manipuladores de eventos podem ser mostrados como encaixes que entram no grão, este último representado por um retângulo. Os eventos são representados por setas ou “plugs”. Figuras que representem um grão disparando um evento em outro grão mostram uma seta saindo do primeiro e chegando a um encaixe do segundo.



Exemplo de grão: Malabarista ou “Juggler”

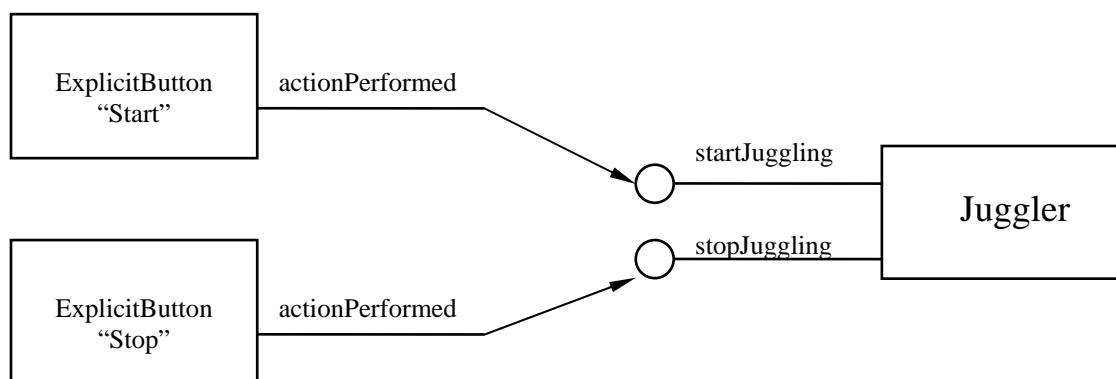
O malabarista é um grão de animação que pode ser iniciado ou interrompido pela conexão de eventos aos manipuladores de evento do malabarista. **ExplicitButton** é um simples grão de botão GUI que permite o texto do botão ser mudado. A cada vez que o botão for pressionado, um evento é disparado.



A figura anterior mostra as possibilidades dos grãos **Juggler** e **ExplicitButton**.

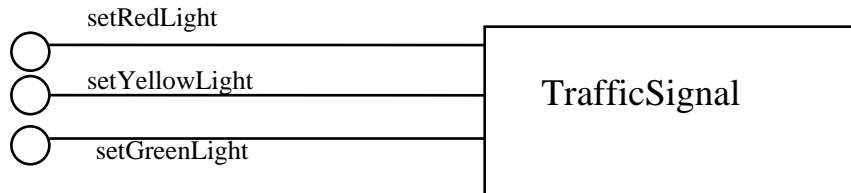
Utilizando **ToolBox** seleciona-se **ExplicitButton**. Em **BeanBox** posiciona-se um botão. Em **Property Sheet** atribui-se o valor “start” ao rótulo do botão. Repete-se o procedimento para um botão “stop”.

Para que o botão “start” inicie o movimento do malabarista e o botão “stop” interrompa o movimento é necessário fazer a conexão entre os botões e as ações a executar. Isto pode ser feito selecionando um botão e agindo nos menus suspensos **Edit|Events|button**. Coloca-se o cursor no grão que vai receber o evento **actionPerformed** e aciona-se o botão do mouse. **BeanBox** exibe a lista dos manipuladores de evento do grão acionado (no caso o **Juggler**). Como existem **startJuggling** e **stopJuggling** faz-se as ligações adequadas.



Exemplo de Java Bean : Sinal de Trânsito

Pretende-se criar um componente para exibir um sinal de trânsito. O gráfico será chamado de **TrafficSignal** e terá três manipuladores de eventos, como se vê a seguir.



Pode-se escrever o código abaixo com a finalidade de implementar um sinal de trânsito.

```
package jdp.beans;

import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class TrafficSignal extends Canvas
{
    public TrafficSignal()
    {
        resize(width, height);
    }

    public void paint(Graphics g)
    {
        int x = spacing;
        int y = spacing;
        int wh = width - (spacing * 2);

        g.setColor(Color.orange);
        g.drawRect(0, 0, width - 1, height - 1);

        // Draw each light

        for (int i = 0; i < 3; i++) {

            switch(i) {
            case RED_LIGHT:
                g.setColor(Color.red);
                break;
            case YELLOW_LIGHT:
                g.setColor(Color.yellow);
                break;
            case GREEN_LIGHT:
                g.setColor(Color.green);
                break;
            }

            // If this is the light that is on, fill
            // it in
        }
    }
}
```

```

        if (i == lightOn) {
            g.fillArc(x, y, wh, wh, 0, 360);
        }
        else {
            g.drawArc(x, y, wh, wh, 0, 360);
        }
        y += (wh + spacing);
    }
}

// Event handling method to turn the light red

public void setRedLight(ActionEvent x)
{
    lightOn = RED_LIGHT;
    repaint();
}

// Event handling method to turn the light yellow

public void setYellowLight(ActionEvent x)
{
    lightOn = YELLOW_LIGHT;
    repaint();
}

// Event handling method to turn the light green

public void setGreenLight(ActionEvent x)
{
    lightOn = GREEN_LIGHT;
    repaint();
}

// Constants that describe the light that will be on

private final static int RED_LIGHT    = 0;
private final static int YELLOW_LIGHT = 1;
private final static int GREEN_LIGHT  = 2;

// Width and height of the traffic canvas. The size of the canvas
// and lights can be controlled by changing the spacing (the space
// between the edge of the canvas and the light) and the total
// width of the canvas.

private int spacing = 5;
private int width   = 30;
private int height  = ((width - (spacing * 2)) * 3) +
                    (spacing * 4);

// Current light that is on

private int lightOn = GREEN_LIGHT;
}

```

Pode-se observar que neste código nada existe que o identifique como um grão de Java. É um simples “widget” Java baseado no objeto **Canvas** e tem alguns métodos públicos para mudar o estado do sinal. Para tornar este código um grão de Java basta lembrar que **Java Beans** são empacotados em arquivos “**Java Archive**”, ou **JAR**. **JAR** é um formato de arquivos para a agregação de múltiplos arquivos que é utilizado para armazenagem de “applets” Java e seus componentes. O armazenamento conjunto desses componentes pode ser de forma comprimida, reduzindo substancialmente o tempo de “download” dos “applets”. Um arquivo **JAR** é um arquivo ZIP que, opcionalmente, pode ter uma tabela de conteúdo chamada de **manifesto**. O **manifesto** descreve o conteúdo do arquivo JAR informando que arquivos de classes são grãos. Não há necessidade da inclusão de arquivos de **manifesto** em arquivos JAR porque todos os arquivos de classes em JAR são considerados grãos.

Para a construção de um arquivo JAR pode-se utilizar o “**Java Developer’s Kit**”, ou **JDK**. O utilitário **jar** executa toda a manutenção e arquivos JAR e pode também criar o arquivo **manifesto**. Seu modo de emprego é da forma

```
jar {ctx} [vfmOm] [arquivo_jar] [arquivo_manifesto] arquivos
```

(**c** de create, **t** de view table of contents e **x** de extract)

As opções são varias. Como exemplo de criação de dois arquivos de classes em um arquivo chamado de **classes.jar** pode-se fazer

```
jar cvf classes.jar Alfa.class Beta.class
```

Depois que tivesse sido compilado o arquivo **TrafficSignal.java**, poder-se-ia criar um arquivo JAR com o comando

```
jar cvfm TrafficSignal.jar manifest.txt jdp\beans\*.class
```

Assim fazendo é criado um arquivo JAR de nome **TrafficSignal.jar** e uma entrada de **manifesto** com o conteúdo de **manifest.txt**. Esta entrada é

```
Name: jdp/beans/TrafficSignal.class
```

```
Java-Bean: True
```

É muito importante deixar uma linha em branco após cada linha contendo ‘Java-Bean: True’, especialmente quando existirem muitos grãos em um arquivo JAR. Todos os Java Beans devem ser parte de um pacote (no caso o pacote foi **jdp/beans**). Uma vez que estejam criados os arquivos JAR devem ser copiados para o repositório de arquivos JAR do **BeanBox**.

Projeto Ponto de Vendas

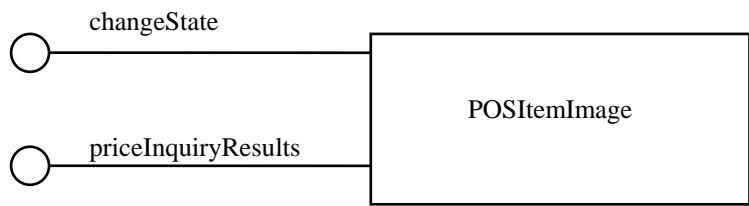
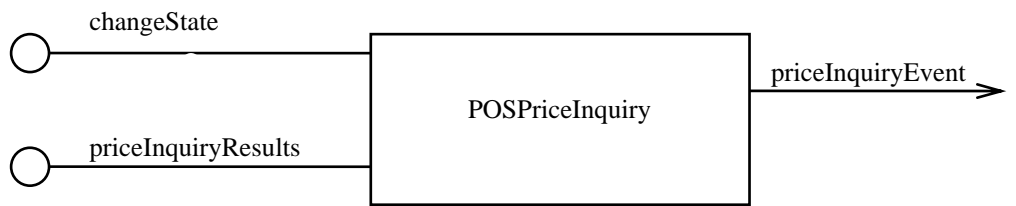
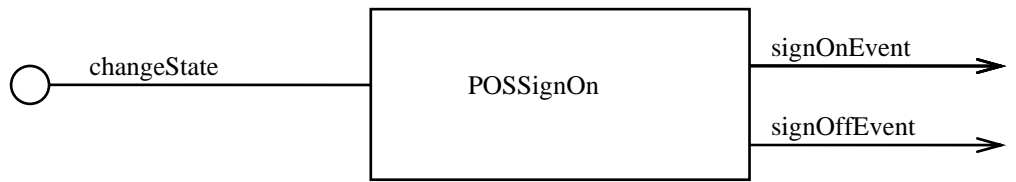
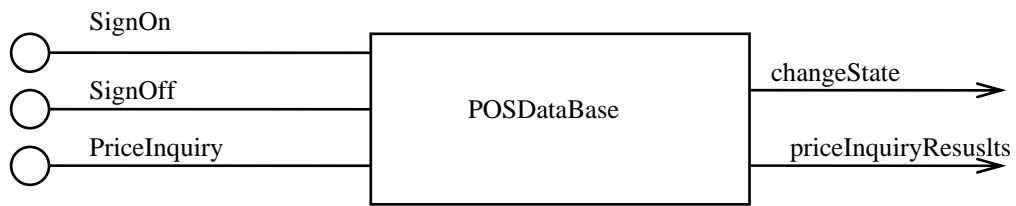
Generalidades

Um exemplo mais elaborado de utilização de grãos de Java é o de Ponto de Vendas, ou “Point of Sale Database”, ou **POS Database**. Este projeto contém quatro grãos:

Grão	Características
POSDatabase	Manuseia todas as tarefas do BD usando JDBC. Exibe também informação de estado sobre o estado atual do sistema. Propriedades editáveis permitem que sejam especificados o driver JDBC e informação sobre a URL.
POSSignOn	Aceita o nome e senha de um usuário para abrir uma sessão no sistema. Com a sessão aberta habilita-se o botão SignOff .
POSPriceInquiry	Aceita um item de código usado para executar uma consulta ao BD. As informações sobre o item são retornadas e exibidas.
POSItemImage	Exibe uma imagem da consulta de preço do item.

O grão POSDatabase utiliza apenas uma tabela, com a composição que se segue:

Nome de coluna	Tipo SQL	Descrição
SKU	Integer	Código de item ou “Stock Keeping Unit”
DESCRIPTION	Varchar	Descrição do item
IMAGE	Varbinary	Imagem binária do item
PRICE	Varvhar	Preço sugerido para venda do item a varejo



O funcionamento do BD de Ponto de Vendas é como se segue:

O usuário entra com sua sigla e senha no grão **POSSignOn** e pressiona o botão **SignOn**. Esta ação envia uma notificação contendo esses dados ao grão **POSDatabase**, estabelecendo uma conexão ao BD especificado no quadro de propriedades do **POSDatabase**. Uma ligação bem sucedida resulta em uma mudança de estado (de desconectado para conectado) e um evento de mudança de estado será enviado para todos os ouvintes registrados (no caso os ouvintes são todos os outros grãos). Isto faz com que o grão **POSSignOn** desabilite o botão **SignOn** e habilite o botão **SignOff**. O grão **POSPriceInquiry** habilita, então, o botão **PriceInquiry**. O usuário pode entrar com um item de código a consultar e pressionar o botão **PriceInquiry**. Esta ação envia um evento **priceInquiry** ao grão **POSDatabase** o que faz com que o BD faça a leitura deste item. Se o item for encontrado o grão **POSDatabase** envia um evento **priceInquiryResults** aos ouvintes de eventos (no caso os grãos **POSPriceInquiry** e **POSItemImage**). Os grãos exibem, então, as informações necessárias sobre o item. Caso o usuário pressione o botão **SignOff** do grão **POSSignOn** a conexão com o BD será fechada, um evento de mudança de estado será enviado e o sistema retornará ao estado inicial.

Os eventos permitem que um grão seja conectado a outro grão, passando informação sobre o evento da fonte do evento (o grão que o criou) ao ouvinte de evento (o grão recebendo a notificação do evento). A fonte do evento mantém registro dos ouvintes de eventos através de um vetor:

```
// Event listener handling

public synchronized void addListener(POSDatabaseEventListener l)
{
    listeners.addElement(l);
}

public synchronized void removeListener(POSDatabaseEventListener l)
{
    listeners.removeElement(l);
}

// Utility objects

private Vector listeners = new Vector();

// Tell whoever is listening that a state change has occurred

public void changeState(int newState)
{
    Vector clonedListeners;
    synchronized(this)
    {
        clonedListeners = (Vector) listeners.clone();
    }

    // If the state is not changing, no need to send out an event

    if (state != newState) {
        state = newState;
        Integer s = new Integer(state);
        for (int i = 0; i < clonedListeners.size(); i++) {

            // Get the listener
            POSDatabaseEventListener l =
```

```

        (POSDatabaseEventListener) clonedListeners.elementAt(i);
        l.changeState(s);
    }
}
}

```

A primeira coisa a fazer é obter uma cópia do vetor. Não se deseja a mudança do vetor pela inclusão ou exclusão de um ouvinte no meio do disparo de um evento e portanto é conveniente trabalhar em uma cópia estável. **BeanBox** guarda informações do método apropriado a invocar no ouvinte do evento. Para isto uma classe de adaptador é criada para agrupá-los. Um exemplo de classe de adaptador é mostrado a seguir:

```

// Automatically generated event hookup file.

package tmp.sun.beanbox;

public class ___Hookup_13fce80237
    Implements jdp.POS.POSDatabaseEventListener, java.io.Serializable {

    public void setTarget(jdp.POS.POSSignOn t) {
        target = t;
    }

    public void changeState (java.lang.Integer arg0) {
        target.changeState (arg0);
    }

private jdp.POS.POSSignOn target;
}

```

Quando se registra um ouvinte de evento, o método **setTarget** da classe de adaptador é chamado com o objeto ouvinte (POSSignOn). Quando **POSDatabase** dispara um evento **changeState** cada ouvinte registrado do método **changeState** é chamado indiretamente usando a classe de adaptador. O método **changeEvent**, do objeto **POSDatabaseEventListener** é chamado e, por sua vez, chama o método apropriado no ouvinte registrado. Neste caso, o nome do método é o mesmo (**changeState**). A classe de adaptador mantém os grãos unidos.

Informação Explícita através de BeanInfo

Uma das formas de descrever algumas facetas de um grão é pela classe **BeanInfo**. Esta classe é obtida adicionando “BeanInfo” ao final do nome da classe do grão. A classe **BeanInfo** é utilizada para especificar os eventos disparados pelos grãos. **BeanBox** pode consultar a classe **BeanInfo** para a informação sobre o evento e modificar apropriadamente o menu **Edit**. A classe **BeanInfo** para **POSDatabase** é do tipo:

```

package jdp.POS;

import java.beans.*;

public class POSDatabaseBeanInfo extends SimpleBeanInfo
{

```

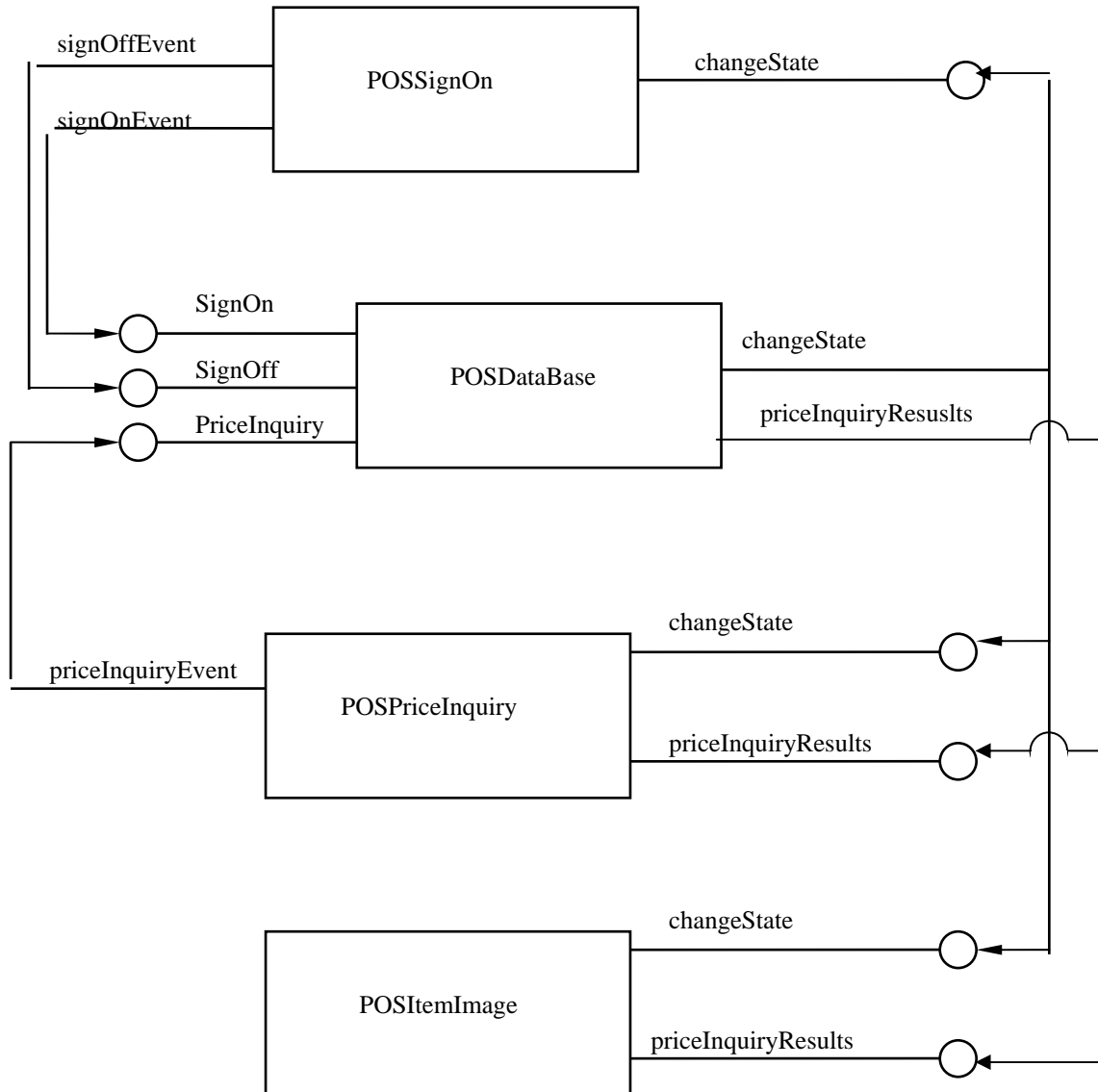
```
// Set the event descriptors for the POSDatabase bean

public EventSetDescriptor[] getEventSetDescriptors()
{
    String[] listenerMethods = {"changeState","priceInquiryResults"};
    EventSetDescriptor event;
    try {
        event = new EventSetDescriptor(POSDatabase.class,
            "Event", POSDatabaseEventListener.class,
            listenerMethods,
            "addListener", "removeListener");
    } catch (IntrospectionException e) {
        throw new Error(e.toString());
    }

    EventSetDescriptor[] events = {event};
    return events;
}
}
```

Juntando tudo

Para colocar tudo junto cria-se o grão **POS SignOn** que aceita sigla e senha de um usuário e habilita o botão **SignOn**. Ao ser acionado este botão ele enviará um evento **SignOn Event** ao grão **POSDataBase**. Este grão faz a conexão com o BD. “O driver” JDBC e a URL de conexão são propriedades editáveis e podem ser especificados na **PropertySheet**. Caso a conexão seja bem sucedida, um evento **changeState** é disparado de volta para o grão **POS SignOn**. Este evento desabilita o botão **SignOn** e habilita o botão **SignOff**. O esquema e o código correspondente são mostrados a seguir:



```
public POSSignOn()
```

```

{
    // Setup our layout manager with 3 rows and 2 columns
    setLayout(new GridLayout(3, 2, 10, 10));

    // Setup the operator ID
    add(new Label("Operator : "));
    operatorField = new TextField();
    add(operatorField);

    // Setup the password
    add(new Label("Password : "));
    passwordField = new TextField();
    add(passwordField);

    // Setup the signon button
    signOnButton = new Button("Sign On");
    signOnButton.addActionListener(this);
    add(signOnButton);

    // Setup the signoff button
    signOffButton = new Button("Sign Off");
    signOffButton.addActionListener(this);
    add(signOffButton);

    // Set the initial state to offline
    changeState(new Integer(POSDefine.OFFLINE));
}

public java.awt.Dimension preferredSize()
{
    return new java.awt.Dimension(150, 90);
}

// Receive an internal state change notification
public void changeState(Integer newState)
{
    int state = newState.intValue();

    if (state == POSDefine.OFFLINE) {
        signOnButton.setEnabled(true);
        signOffButton.setEnabled(false);
        operatorField.setText("");
        operatorField.setEditable(true);
        passwordField.setText("");
        passwordField.setEditable(true);
    }
    else {
        signOnButton.setEnabled(false);
        signOffButton.setEnabled(true);
        operatorField.setEditable(false);
        passwordField.setText("");
        passwordField.setEditable(false);
    }
}

```

```

    }
}

// Event listener handling

public synchronized void addListener(POSSignOnEventListener l)
{
    listeners.addElement(l);
}

public synchronized void removeListener(POSSignOnEventListener l)
{
    listeners.removeElement(l);
}

// ActionListener implementation

public void actionPerformed(ActionEvent e)
{
    // Get the source of the action

    Object source = e.getSource();

    // Perform some action depending upon the source

    if (source == signOnButton) {
        Vector clonedListeners;
        synchronized(this) {
            clonedListeners = (Vector) listeners.clone();
        }

        POSSignOnObject signOnObject = new POSSignOnObject();
        signOnObject.setOperator(operatorField.getText());
        signOnObject.setPassword(passwordField.getText());

        for (int i = 0; i < clonedListeners.size(); i++) {

            // Get the listener
            POSSignOnEventListener l =
                (POSSignOnEventListener) clonedListeners.elementAt(i);
            l.signOnEvent(signOnObject);

        }
    }

    if (source == signOffButton) {
        Vector clonedListeners;
        synchronized(this) {
            clonedListeners = (Vector) listeners.clone();
        }

        for (int i = 0; i < clonedListeners.size(); i++) {

            // Get the listener
            POSSignOnEventListener l =
                (POSSignOnEventListener) clonedListeners.elementAt(i);
            l.signOffEvent(null);
        }
    }
}

```

```

        }
    }

}

// Layout fields

private TextField operatorField;
private TextField passwordField;
private Button    signOnButton;
private Button    signOffButton;

// Utility objects

private Vector listeners = new Vector();
}

```

O grão **POSSignOn** estende **Panel** e estava sendo usado **GridLayout** para posicionar os componentes da GUI. O botão **SignOnButton** e **SignOffButton** estão ligados ao método **actionPerformed** quando pressionados. A partir do método **actionPerformed** dispara-se o evento apropriado dependendo do botão pressionado. Existe também o método que é um ouvinte de evento. O método **changeState** cuida da habilitação e desabilitação de componentes na conexão e desconexão de usuários. O evento **SignOnEvent**, ao ser disparado para quem estiver ouvindo, será invocado o método **signOn** do grão **POSDatabase**, a seguir exibido.

```

public void SignOn(POSSignOnObject signOnObject)
{
    boolean failed = false;

    // No action if we're already online
    if (state == POSDefine.ONLINE) {
        return;
    }

    try {

        // Connect to the driver

        Driver d = (Driver) Class.forName(driverName).newInstance();
        con = DriverManager.getConnection(connectionURL,
                                       signOnObject.getOperator(),
                                       signOnObject.getPassword());

    }
    catch (Exception ex) {
        setError(ex.getMessage());

        // Dump the stack for debugging purposes
        ex.printStackTrace();

        failed = true;
    }
}

```

```

// If everything went OK, change our internal
// state to ONLINE, also notifying all other
// beans who are listening

if (!failed) {

    // Set the operator name

    operatorLabel.setText(POSDefine.OPERATOR +
        signOnObject.getOperator());
    operatorLabel.repaint();
    setOnline();
}

repaint();
}

```

Para fazer a conexão o objeto **POSSignOn**, quando recebe um evento disparador, recebe também sigla e senha do usuário. A conexão é tentada e, se houver uma falha o estado da linha no grão **POSDatabase** mostra um erro. Se a conexão obtém sucesso é disparado um evento **changeState** para notificar os objetos ouvintes que existe uma conexão no ar. A desconexão é feita encerrando a conexão e disparando um evento notificando a todos os ouvintes de eventos o término da conexão.

```

public void SignOff()
{
    boolean failed = false;

    try {
        if (con != null) {
            con.close();
            con = null;
        }
    }
    catch (Exception ex) {
        setError(ex.getMessage());
        failed = true;
    }

    // Log off the database

    setOffline();
    repaint();
}

```

Consulta de preços

O grão **POSPriceInquiry** permite a entrada de um código de item e o disparo de um evento **PriceInquiry** para cada item. O grão **POSDatabase** está na escuta e o método **PriceInquiry** é chamado.

```

public void PriceInquiry(POSItemObject itemObject)
{

```

```

boolean failed = false;
java.sql.Statement stmt = null;
ResultSet rs = null;
String item;
POSItemObject o = null;

item = itemObject.getItemCode();

if ((item == null) ||
    (item.length() == 0)) {
    setError("Item code required for price inquiry");
    repaint();
    return;
}

try {
    // Create a statement object

    stmt = con.createStatement();

    // Execute the query

    rs = stmt.executeQuery("select SKU,DESCRIPTION,PRICE,IMAGE "
        + "from ITEM " +
        "where SKU=" + item);

    // Get the results

    if (!rs.next()) {
        setError("Item '" + item + "' not found");
        failed = true;
    }
    else {
        o = new POSItemObject();
        o.setItemCode(rs.getString(1));
        o.setItemDescription(rs.getString(2));
        o.setItemPrice(rs.getString(3));

        // We'll handle the image by getting the column as
        // an InputStream then reading it into a byte array.

        java.io.InputStream is = rs.getBinaryStream(4);
        byte b[] = new byte[is.available()];
        is.read(b);
        o.setItemImage(b);
    }
}

catch (Exception ex) {
    setError(ex.getMessage());

    // Dump the stack for debugging purposes
    ex.printStackTrace();

    failed = true;
}

// If everything went OK, send the item information

```

```

        // out to whoever is expecting it

        if (!failed) {
            priceInquiryResults(o);
            setOnline();
        }

        repaint();
    }
}

```

Faz-se a consistência dos dados, um novo objeto **Statement** é criado e um comando SQL SELECT, é formado. Caso este comando seja executado e um **ResultSet** retornado, visualiza-se a primeira linha dos dados. Esses dados são enviados juntamente com o evento **PriceInquiryResult**.

Trabalhando com Imagens

A exibição da imagem do item é feita pelo grão **POSItemImage**. Como Java não exibe imagens diretamente a partir de “arrays” de bytes, o que se pode fazer é gravar os dados binários em um arquivo e depois usar classes JDK que carregam uma imagem do disco e a exibem em um “canvas”, como se segue.

```

// Receive a price inquiry result notification

public void priceInquiryResults(POSItemObject o)
{
    // Here's where things get ugly. Currently, there is no way
    // (using the JDK) to display an image from a byte array.
    // Note that there are vendors who supply classes do perform this
    // task, as well as 'freeware' classes such as GifImage. We'll
    // create a new file from the InputStream and load the Image
    // from the newly created file.

    String loadName = null;

    deleteFile(lastFileName);

    try {
        lastFileName = tempFileName + "-" + o.getItemCode();
        FileOutputStream outputStream = new
FileOutputStream(lastFileName);
        byte b[] = o.getItemImage();
        outputStream.write(b, 0, b.length);
        outputStream.close();

        // If we got here, everything worked OK. Set the name
        // of the file to load

        loadName = lastFileName;
    }
    catch (Exception ex) {
        // Dump the stack for debugging purposes
        ex.printStackTrace();
    }

    loadFile(loadName);
}

```

```
    repaint();  
}
```