Módulo 9

Banco de Dados

Lição 6

Java Database Connectivity (JDBC)

Versão 1.0 - Fev/2009

Autor
Ma. Rowena C. Solamo

Equipe
Rommel Feria
Rick Hillegas
John Paul Petines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/60/relnotes.htm

Java™ DB System Requirements

Java™ DB is supported on the Solaris, Linux and Windows operating systems and Sun Java 1.4 or later.

Colaboradores que auxiliaram no processo de tradução e revisão

Aécio Júnior
Alberto Ivo da Costa Vieira
Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
Angelo de Oliveira
Aurélio Soares Neto
Bruno da Silva Bonfim
Carlos Fernando Gonçalves
Carlos Hilner Ferreira Costa
Daniel Noto Paiva
Daniel Wildt
Denis Mitsuo Nakasaki
Fábio Antonio Ferreira
Givailson de Souza Neves
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Vianney Barrozo Costa
José Francisco Baronio da Costa
Kleberth Bezerra Galvão dos Santos
Luiz Fernandes de Oliveira Junior
Maria Carolina Ferreira da Silva
Maricy Caregnato
Mauricio da Silva Marinho
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Sergio Terzella
Thiago Magela Rodrigues Dias

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

JDBC é o acrônimo para Java Database Connectivity. Trata-se de uma API Java que permite a programas escritos em Java executarem instruções SQL. Esta API permite que programas Java possam interagir com qualquer bando de dados compatível com SQL. Uma vez que todos os sistemas gerenciadores de bancos de dados relacionais (DBMSs) suportam SQL e pelo fato de Java rodar na maioria das plataformas, com JDBC é possível escrever uma simples aplicação que rode em diferentes plataformas e interaja com diferentes DBMSs. Esta lição compreende os seguintes tópicos:

- JDBC Design Pattern
- Implementando JDBC Design Patterns
- Programação de Bancos de Dados server-side
  - Stored Procedures com JDBC
  - Stored Functions com JDBC

Ao final desta lição, o estudante será capaz de:

- Utilizar a Java Database Connectivity (JDBC)
- Entender como utilizar a JDBC Design Pattern
- Utilizado a API Java que permite programas Java a execução de comandos SQL
- Implementar uma uma aplicação de inventário que manipula registros da tabela de inventário
- Utilizar a JavaDB como banco de dados
- Como criar procedimentos armazenados e funções como programações de servidor de banco de dados

2. JDBC Design Pattern[1]

Esta sessão discute o padrão de uso do mecanismo de persistencia escolhido para o Sistema de Gerenciamento de Bancos de Dados Relacionais (RDBMS) que é o Java Database Connectivity (JDBC) Design Pattern. Existem duas visões para este pattern, chamadas: ***static view*** e ***dynamic*** view.

2.1. Visão Estática do Design Pattern de Persistência

A visão estática é um modelo de objeto do pattern. Isto é ilustrado usando um diagrama de classes. A Figura 1 mostra o design pattern para as classes persistentes.

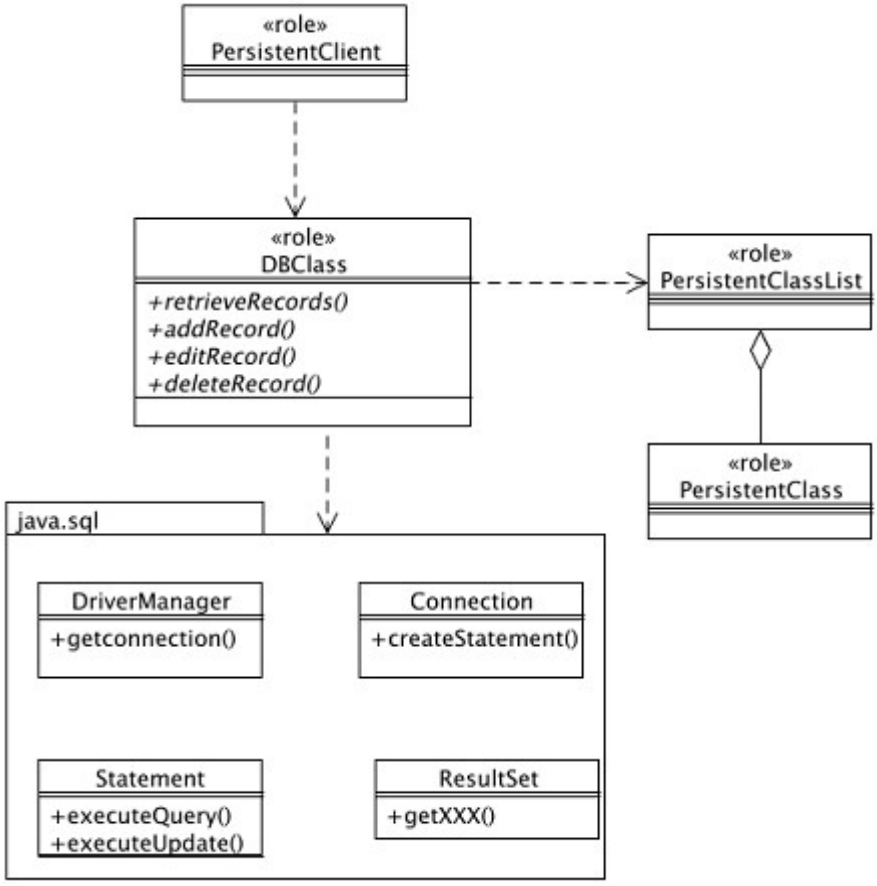

Figura 1: JDBC Design Pattern Visão Estática

Tabela 1 mostra as classes que são definidas pelo programador.

Classe	*Descrição*
`PersistentClient`	É uma classe de requisição de dados do banco de dados. Normalmente, são classes de controle perguntando algo a partir de uma entity class. Ela trabalha como uma **`DBClass.`**
`DBClass`	É uma classe responsável pela comunicação com o banco de dados. Trabalha em conjunto com as classes do pacote `java.sql`.
`PersistentClassList`	É uma classe usada para retornar um conjunto de objetos persistentes com resultado de uma consulta ao banco de dados. Um registro é equivalente a uma **`PersistentClass`** nesta lista.

1 A sessão do JDBC Design Pattern foi copiada do JEDI-Software Engineering Courseware. Seu uso é de conhecimento do autor.

Classe	*Descrição*
`PersistentClass`	É uma classe que mapeia um registro no banco de dados.

Tabela 1: JDBC Design Pattern Classes

A **DBClass** é responsável por tornar outras instâncias de classes persistentes. Entendemos por isso como o mapeamento OO-para-RDBMS. Esta classe estebelece uma interface com o RDBMS. **Toda classe que precisar ser persistida deverá ter uma DBClass correspondente!**

Tabela 2 mostra as classes que estão definidas no pacote java.sql.

Classes	Método	Descrição do Método
Class	forName()	Carrega o driver apropriado para o banco de dados.
DriverManager - É uma classe que permite a manipulação do driver JDBC que foi carregado.	getConnection()	Estabelece uma conexão com o banco de dados alvo. Quando conectado com sucesso, este método retorna um objeto Connection. Caso contrário, uma SQLException é levantada.
Connection É uma classe que representa uma conexão para o banco de dados.	createStatement()	Retorna um objeto Statement.
	close()	Fecha uma conexão.
Statement É uma classe que representa uma instrução SQL. **PreparedStatement** e **CallableStatement** são usadas para executar stored procedures e functions.	executeQuery()	Requisita o banco para executar uma instrução SELECT-statement (query). Retorna um objeto ResultSet contendo todas as linhas que satisfazem a SELECT-statement. Caso contrário, retorna null.
	executeUpdate()	Requisita o banco de dados para executar tanto uma instrução INSERT, UPDATE ou DELETE. Pode também exectar Data Definition Language como instruções ALTER-statements.
	close()	Fecha uma statement.
ResultSet É uma classe que representa o conjunto de resultados, i.e., o conjunto de registros retornados pela consulta. Existem outros métodos getXXX() para diferentes tipos de dados primitivos usados em Java.	hasNext()	Retorna um valor booleano. Se ainda existirem registros no ResultSet, retorna true. Caso contrário retorna false.
	next()	Retorna o próximo registro do ResultSet.
	getString()	Recupera o valor String da coluna especificada assim como referenciado pelo número que representa a posição do valor dentro do registro.
	getInt()	Recupera o valor inteiro da coluna especificada assim como referenciado pelo número que representa a posição do valor dentro do registro.
	getLong()	Recupera o valor inteiro longo da coluna especificada assim como referenciado pelo número que representa a posição do valor dentro do registro.
	close()	Fecha o resultset.

Tabela 2: Classes comuns do pacote java.sql

Note que as classes Connection, Statement, e ResultSet possuem um método close(). Isto é bom

para deixar claro quem está sendo fechado. Estas classes são alocadas nos recursos do sistema quando são criadas e usadas. Para liberar recursos do sistema, precisamos explicitamente fechá-las, invocando o método close().

O pacote java.sql é usado porque contem classes e interfaces para manipulação de bancos de dados relacionais em Java.

2.2. Visão Dinâmica do Padrão de Projeto de Persistência

A visão dinâmica do Padrão de Projeto de Persistência mostra como as classes a partir da visão estática interagem umas com as outras. O diagrama de seqüência é usado para ilustrar este comportamento dinâmico. Existem alguns comportamentos dinâmicos que podem ser vistos neste padrão, especificamente, são eles: ***JDBC Initialization***, ***JDBC Create***, ***JDBC Read***, ***JDBC Update*** e ***JDBC Delete***.

1. ***JDBC Initialization***. A inicialização deve ocorrer antes que qualquer classe persistente possa ser acessada. A principal necessidade é estabelecer conexão com o RDBMS. Isto envolve o seguinte:
   - Carregar apropriadamente o *driver*
   - Conexão com o banco de dados

O diagrama de seqüência da inicialização do JDBC é exibido na Figura 2.

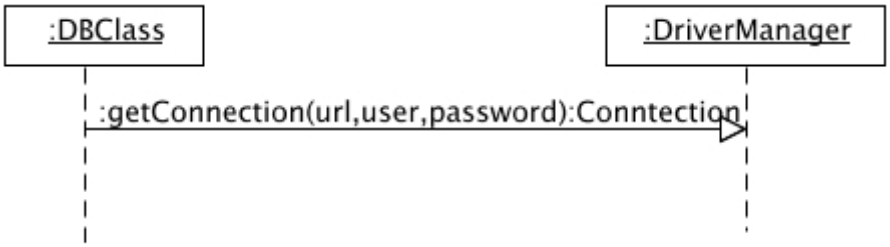

Figura 2: Inicialização do JDBC

A DBClass carrega o *driver* apropriado chamando getConnection(url, user, password). Este método estabelece uma conexão com o banco de dados informado na URL de conexão. O DriverManager seleciona o *driver* apropriado a partir de um conjunto de drivers JDBC registrados.

2. ***JDBC Create***. Este comportamento cria um registro. Executa a instrução SQL INSERT. Ele considera que a conexão já esteja estabelecida. O diagrama de seqüência é mostrado na Figure 3.

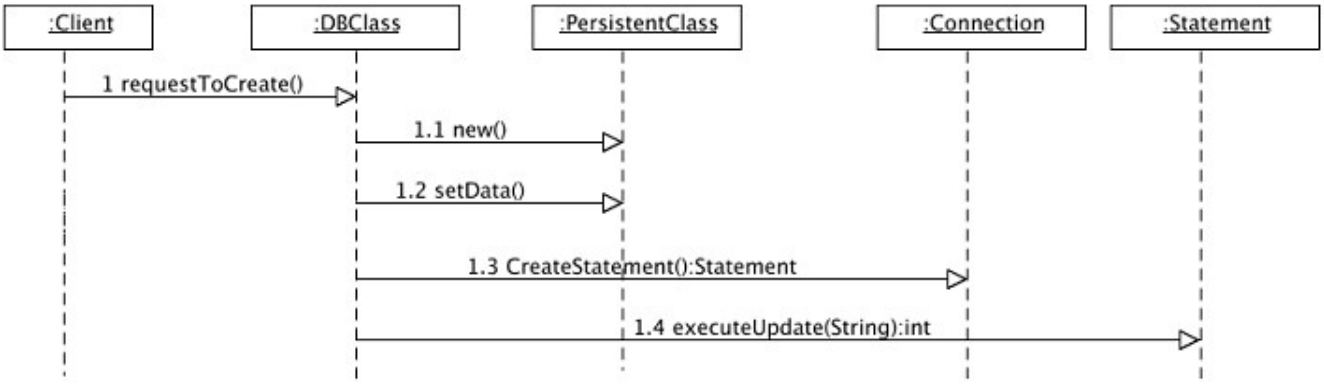

Figure 3: JDBC Create

Descrição do Fluxo do Diagrama de Seqüência:

- A **PersistencyClient pergunta solicita à DBClass que crie uma nova classe**.
- A **DBClass** cria uma nova instância de **PersistentClass (new())** e atribui valores para a **PersistentClass.**

- A **DBClass,** então, cria uma nova declaração usando **createStatement()** de **connection.** Normalmente, trata-se de uma instrução INSERT em SQL.
- A **Statement** é executada por meio do método **executeUpdate(String):int**. Um registro é inserido no banco de dados.

3. ***JDBC Read***. Este comportamento recupera registros do banco de dados. Ele executa a instrução SELECT. Ele também presume que uma conexão com o banco já esteja estabelecida. O diagrama de seqüência é mostrado na Figura 4.

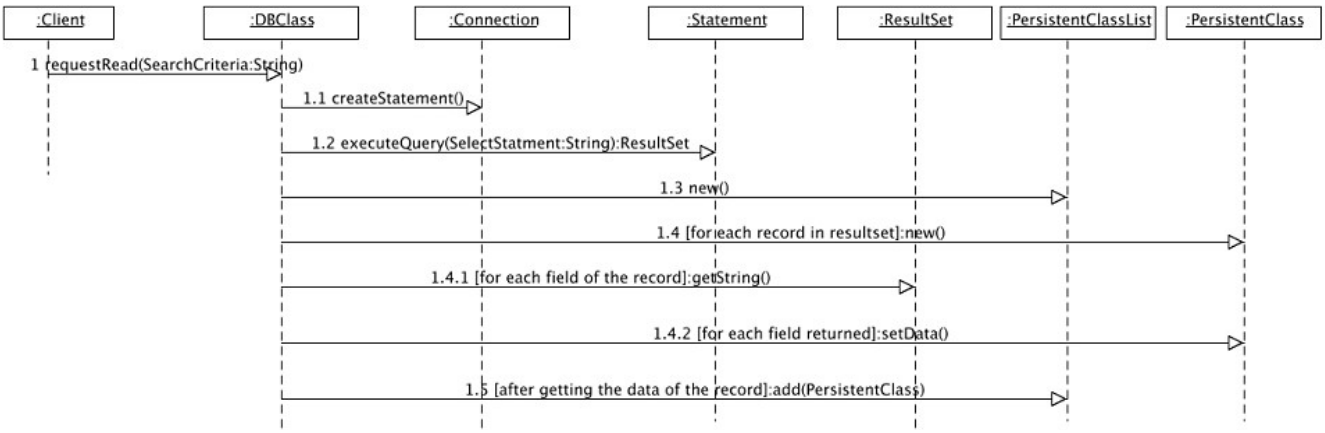

Figura 4: JDBC Read

- A **PersistencyClient** solicita à **DBClass** que recupere registros do banco de dados. A string **SearchCriteria** qualifica que registros serão retornados.
- A **DBClass** cria uma declaração SELECT **Statement** usando o método **createStatement()** de **Connection.**
- A **Statement** é executada via **executeQuery()** e retorna um **ResultSet.**
- A **DBClass** instância uma **PersistentClassList** para manter os registros que estão no **ResultSet.**
- Para cada registro no **ResultSet,** a **DBClass** instancia uma **PersistentClass.**
- Para cada campo no registro, atribui o valor do campo (**getString()**) ao atributo apropriado na **PersistentClass (setData()).**
- Após obter todos os dados do registro e mapeá-los para os atributos da **PersistentClass,** adiciona a **PersistentClass** à **PersistentClassList.**

4. ***JDBC Update***. Executa a instrução UPDATE de SQL. Modifica os valores de um registro existente no banco de dados. Assume que uma conexão já esteja estabelecida. O diagrama de seqüência é mostrado na Figura 5.

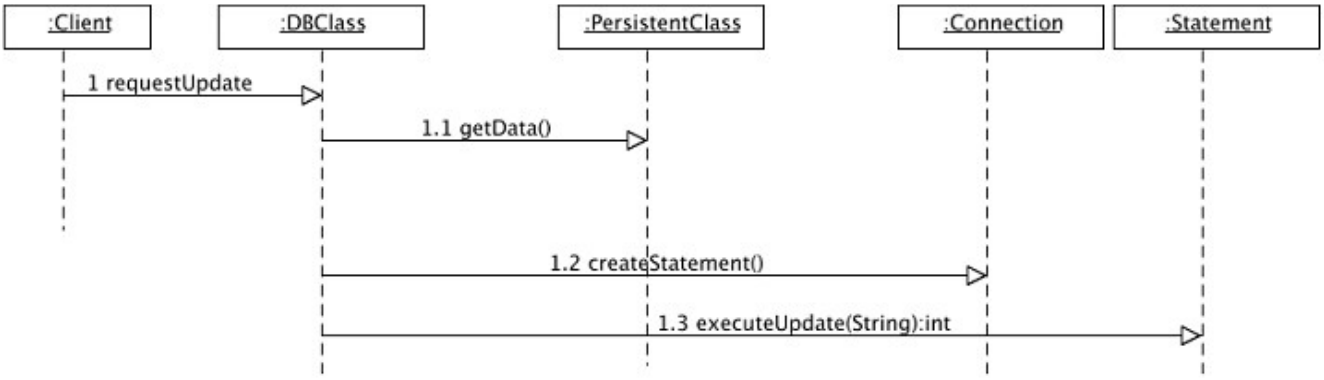

Figura 5: JDBC Update

- **PersistencyClient** solicita à **DBClass** que atualize um registro.
- A **DBClass** recupera o dado apropriado da **PersistentClass.** A **PersistentClass** deve prover uma rotina de acesso para todos os dados persistentes que a **DBClass** precisar. Isto irá garantir acesso externo a certos atributos persistentes que deveriam ser privados. Esta é uma prática para que você mantenha o conhecimento sobre persistência fora da classe que encapsula os atributos persistentes.
- A **Connection** irá criar uma instrução UPDATE.
- Uma vez que a **Statement** é construída, a instrução UPDATE é executada e o banco de dados é atualizado com os novos dados da classe, vale dizer que o estado da classe será atalizado no banco de dados.

5. ***JDBC Delete***. Executa uma instrução DELETE em SQL. Esta instrução elimina os registros em um banco de dados. Presume que uma conexão com o banco de dados já esteja estabelecida. O diagrama de seqüência é mostrado na Figura 6.

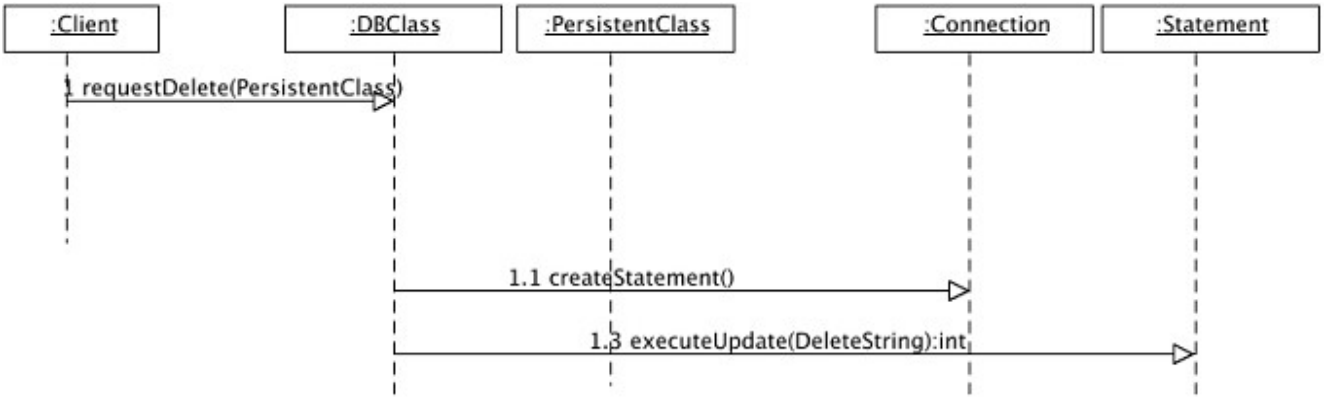

Figura 6: JDBC Delete

- A **PersistencyClient** solicita à **DBClass** que exclua o registro.
- A **DBClass** cria uma instrução DELETE e a executa por meio do método **executeUpdate()**.

3. Implementando JDBC Design Pattern

Esta sessão mostra como escrever código Java que conecta e solicita serviços de um banco de dados, obedecendo o padrão JDBC Design Pattern. Para trabalhar com JavaDB no NetBeans, adicione as bibliotecas `derby.jar`, `derbyclient.jar` e `derbytools.jar` como bilbiotecas do projeto. A Figura 7 mostra os arquivos jar com parte do projeto atual (Organicshop).

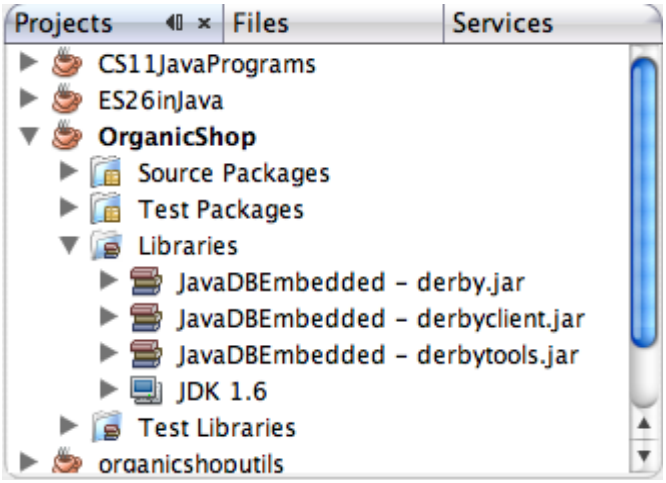

Figura 7: Bibliotecas JavaDB/Derby

O pacote java.sql é usado porque contém classes e interfaces para manipulação de bancos de dados relacionais em Java.

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

A instrução import especifica que queremos usar as classes e interfaces definidas no pacote java.sql. A Tabela 2 lista as classes comuns que são usadas pelo programa Java.

O exemplo que será usado é o The Organic Shop. Vamos assumir que uma aplicação Java irá prover o inventário de uma loja em particular. O diagrama de classes está representado pela Figura 8.

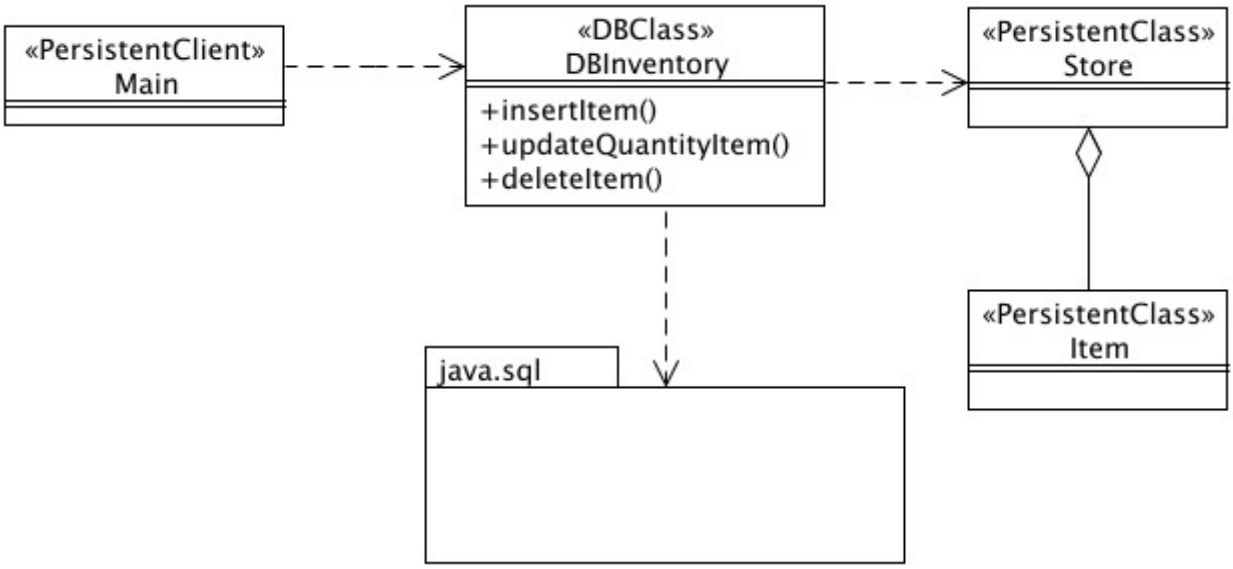

Figura 8: Diagrama de Classe do Inventory By Story Java

A Tabela 3 descreve cada classe do Diagrama de Classes Inventory By Store Java segundo as orientações do padrão JDBC Design Pattern.

Classe	Método	Descrição do Método
Main É o cliente que solicita serviços ao banco de dados. Usa a classe `Store` como um meio para manter os dados para ler e escrever no banco de dados.		
DBInventory É a **DBClass** responsável por ler e escrever dados persistentes do inventário no banco de dados. Utiliza a classe Store para ler e escrever dados no banco de dados.	getInventoryByStore()	Retorna um objeto Store que representa a loja e seus itens de inventário.
	saveNewItem()	Salva um novo item vendido pela loja.
	updateCount()	Atualiza a quantidade em estoque de um item em particular.
	deleteAnItem()	Exclui um item vendido pela loja.
	establishConnection()	Método privado que estabelece uma conexão com o banco de dados.
	closeConnection()	Método privado que fecha uma conexão estabelecida com o banco de dados.
Store É a **classe persistente** que contém o inventário de uma loja. Um de seus atributos é uma lista de itens.	toString()	Retorna uma string que contem o número da loja e o nome com o inventário da loja, i.e., todos os itens com suas respectivas quantidades em estoque e nível de operação.
Item É uma **classe persistente** que contém o item que é vendido, sua quantidade e nível operacional.	toString()	Retorna uma string que contém o código do item, descrição do item, quantidade em estoque e nível de operação de um item vendido na loja.

Tabela 3: Descrição das Classes de Inventory By Store

3.1. Estabelecendo e Fechando uma Conexão

A classe Main é o Persistent Client que solicita serviços para o banco de dados. Ele se comunica com a classe DBInventory. O código abaixo mostra a declaração de seus atributos e os métodos establishConnection() e closeConnection(). Observe que os atributos da classe DBInventory incluem objetos das classes Connection, Statement e ResultSet.

Antes que possamos solicitar qualquer serviço do banco de dados, uma conexão precisa ser estabelecida. O método establishConnection() nos dá o código para conexão com o banco de dados. As linhas seguintes especificam a URL (Uniform Resource Locator) do banco de dados que ajuda o programa a localizar o banco de dados, username e password. A URL especifica o protocolo para comunicação **(jdbc),** o subprotocolo (**derby**) e o nome do banco de dados **(organicshop).** O username e a password também são especificados para autenticação no banco de dados; no exemplo, o username é "nbuser" e a password é "nbuser".

```
String url = "jdbc:derby://localhost:1527/organicshop";
String username = "nbuser";
String password = "nbuser";
```

A classe que define o *driver* para o banco de dados deve estar carregada antes de o programa conectar-se ao banco de dados. As linhas seguintes carregam o *driver*. Neste exemplo, o *driver*

embutido do JavaDB é carregado para permitir que qualquer programa Java tenha acesso a um banco de dados JavaDB.

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

Para estabelecer uma conexão, um objeto do tipo *Connection* é criado. Ele gerencia a conexão entre o programa Java e o banco de dados. Ele também provê suporte para a execução de instruções SQL. No exemplo, a seguinte instrução provê esta conexão.

```
this.theConnection = DriverManager.getConnection(url, username, password);
```

Note que as classes *Class* e *DriverManager* estão dentro de um bloco try-catch. Se um erro ocorrer, uma exceção do tipo *ClassNotFoundException* é disparada para a instrução *Class.forName*() quando o *driver* não puder ser carregado ou localizado, enquanto uma *SQLException* é levantada quando uma conexão não puder ser estabelecida.

```
package organicshop.database.dbclass;

import java.sql.*;
import java.util.List;
import java.util.ArrayList;
import organicshop.database.persistent.Store;
import organicshop.database.persistent.Item;

public class DBInventory {
  private Store theStore;
  private int storeNumber;
  private Connection theConnection;
  private Statement theStatement;
  private ResultSet theResultSet;

  private void establishConnection(){
    String url = "jdbc:derby://localhost:1527/organicshop";
    String username = "nbuser";
    String password = "nbuser";
    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
      this.theConnection =
        DriverManager.getConnection(url,username,password);
    } catch (ClassNotFoundException cnfx){
      System.err.println("Organic Shop: Cannot Load Driver");
      cnfx.printStackTrace();
      System.exit(1);
    } catch(SQLException sqlx){
      System.err.println(
        "Organic Shop: Unable to connection to the database");
      sqlx.printStackTrace();
      System.exit(1);
    }
  }
  private void closeConnection(){
    try{
      System.out.println(
        "Organic Shop: SQL processing done, and closing " +
        "the connection.");
      this.theConnection.close();
    } catch (SQLException sqlx){
      System.err.println(
        "Organic Shop: Unable to close the connection");
      sqlx.printStackTrace();
      System.exit(1);
    }
  }
}
```

É claro, após usar o banco de dados, devemos fechar a conexão. O método *closeConnection*() faz

isso. Observe que *this.theConnection.close*() necessita estar dentro de um bloco try-catch. Dispara uma *SQLException* quando uma conexão não puder ser fechada.

```
public Store getInventoryByStore(int theStoreNumber){
  this.storeNumber = theStoreNumber;
  this.establishConnection();
  this.populateStoreData();
  this.populateItemData();
  this.closeConnection();
  return this.theStore;
}
private void populateStoreData(){
  String selectStmt = "SELECT name FROM store WHERE number = " +
    this.storeNumber;
  try{
    this.theStatement = this.theConnection.createStatement();
    this.theResultSet = this.theStatement.executeQuery(selectStmt);
    if (this.theResultSet != null){
      this.theStore = new Store();
      this.theStore.setNumber(this.storeNumber);
      if (this.theResultSet.next()){
        this.theStore.setName(this.theResultSet.getString(1));
      } else {
        this.theStore.setName("No name.");
      }
    } else {
      System.out.println("Organic Shop: No records retrieved.");
    }
    this.theStatement.close();
  } catch (SQLException sqlx){
    sqlx.printStackTrace();
  }
}
private void populateItemData(){
  List<Item> theItemList = null;
  String selectStmt = "SELECT item.code, item.description, quantity," +
    "op_level ";
  selectStmt = selectStmt + "FROM item, inventory ";
  selectStmt = selectStmt + "WHERE inventory.item_code = item.code ";
  selectStmt = selectStmt + "AND inventory.store_no = " + this.storeNumber;
  try {
    this.theStatement = this.theConnection.createStatement();
    this.theResultSet = this.theStatement.executeQuery(selectStmt);
    if (this.theResultSet != null){
      theItemList = new ArrayList<Item>();
      while(this.theResultSet.next()){
        Item theItem = new Item();
        theItem.setCode(this.theResultSet.getInt(1));
        theItem.setName(this.theResultSet.getString(2));
        theItem.setQuantity(this.theResultSet.getInt(3));
        theItem.setLevel(this.theResultSet.getInt(4));
        theItemList.add(theItem);
      }
    }
    this.theStore.setItemList(theItemList);
    this.theStatement.close();
  } catch (SQLException sqlx){
    sqlx.printStackTrace();
  }
}
```

3.2. Consultando o Database

Assumiremos que queremos ver o inventário de uma loja em particular. Vamos consultar o banco de dados Organic Shop para obter esta informação. O código mostra-nos o código de como ler ou

recuperar dados de um banco de dados. Usaremos o *theStore*(Objeto Store) como o objeto de transferência de dados, i.e., o objeto que será usado para manter os dados que resultarão da consulta ao banco de dados. Para ler ou recuperar dados do banco de dados, o método *getInventoryByStore*() é invocado. Ele retorna um objeto *Store* que contém informações da loja e o inventário de seus itens vendidos. Ele chama dois outros métodos para realizar a consulto ao banco de dados e popular a classe persistente, o objeto *theStore*. A *string selectStmt* é usada para definir instruções SELECT que especifica que registros serão recuperados do banco de dados. O objeto *theStatement* é usado para consultar o banco de dados e é instanciado pelo método *createStatement*() do objeto *theConnection*. O método executeQuery() do objeto theStatement é usado para executar a instrução SELECT como especificado na *string selectStmt*. O resultado da consulta é colocado e referenciado em *theResultSet* que é um objeto do tipo *ResultSet*. Ele contém os registros recuperados pelo banco de dados.

O objeto *ResultSet* possui métodos que retornam dados da coluna da tabela. Se o valor da coluna é do tipo VARCHAR, então usamos o método *getString*(). Se o valor da coluna é um inteiro, então usamos o método *getInt*(). No exemplo abaixo, o atributo name so objeto *theStore* é atribuído ao valor da primeira coluna da linha atual referenciada em *theResultSet*.

```
this.theStore.setName(this.theResultSet.getString(1));
```

Uma vez que os valores do objeto *theStore* tenham sido atribuídos, são passados de volta a classe que fez a chamada. Neste caso, a classe *Main*. O código em *Main* que instancia um *theInventory* (Objeto do tipo *DBInventory*, e chama o método *getInventoryByStore*(). Ele espera retornar um objeto *Store* referenciado pela variável *theStore*.

```
// The code is written in Main.java
DBInventory theInventory = new DBInventory();
Store theStore;
Item theItem;
theStore = theInventory.getInventoryByStore(10);
if (theStore != null){
  System.out.println(theStore.toString());
} else {
  System.out.println("The Store is null.");
}
```

3.3. Inserindo um Registro

Vamos assumir que aloja decidiu vender um novo item. O inventário deste item deve ser mantido no banco de dados. O código Java que insere um registro no banco de dados. Dois métodos são definidos na classe DBInventory; chamados, saveNewItem() e insertItem(). O primeiro método é usado pelo cliente persistente para invocar a inclusão. Todavia, o segundo método encarrega-se da atual inclusão.

Dentro do método insertItem(), a string insertStmt é usada para manter a instrução atual INSERT-statement com o conjunto de valores apropriados. O processo de consulta é similar, usamos o objeto theConnection para criar um objeto theStatement. Ao invés de usar o método executeQuery(), usamos o método executeUpdate(). Por último, to código é cercado por um bloco try-and-catch para que no caso de falha no processo, estejamos prontos para tratar uma possível exceção do tipo SQLException gerada.

```
public void saveNewItem(Item theItem1){
  this.theItem = theItem1;
  this.establishConnection();
  this.insertItem();
  this.closeConnection();
}
private void insertItem(){
  String insertStmt = "INSERT INTO inventory VALUES (";
  insertStmt = insertStmt + this.theItem.getStoreNumber() + ",";
  insertStmt = insertStmt + this.theItem.getCode() + ",";
  insertStmt = insertStmt + this.theItem.getQuantity() + ",";
  insertStmt = insertStmt + this.theItem.getLevel() + ")";
  try{
```

```
        this.theStatement = this.theConnection.createStatement();
        this.theStatement.executeUpdate(insertStmt);
        System.out.println("Organic Shop: Row has been inserted.");
        this.theStatement.close();
      } catch (SQLException sqlx){
        sqlx.printStackTrace();
      }
    }
```

O código a seguir dentro de **Main** (o cliente persistente) mostra como chamar o método *saveNewItem*().

```
DBInventory theInventory = new DBInventory();
Item theItem = new Item();

// Enviar novos valores ao item
theItem.setStoreNumber(10);
theItem.setCode(1002);
theItem.setQuantity(1500);
theItem.setLevel(500);

// Chamar o método saveNewItem enviando theItem para inserir
theInventory.saveNewItem(theItem);
theStore = theInventory.getInventoryByStore(10);
if (theStore != null){
   System.out.println(theStore.toString());
} else {
   System.out.println("The Store is null.");
}
```

3.4. Atualizando um Registro

Vamos supor que precisamos atualizar o valor do atributo quantidade do novo registro que acabamos de inserir. O código abaixo demonstra como atualizar o valor. Dois métodos estão definidos na classe DBInventory; chamados: updateQuantityItem() e setQuantity(). O segundo método executa uma instrução UPDATE. Note que a estrutura é a mesma do processo de insersão, exceto pela instrução INSERT, agora especificamos uma instrução UPDATE.

A string updateStmt declarada no método setQuantity() especifica a instrução UPDATE usada para modificar o valor do atributo quantidade.

```
//This code is written in DBInventory.java.
public void updateQuantityItem(Item theItem1){
   this.theItem = theItem1;
   this.establishConnection();
   this.setQuantity();
   this.closeConnection();
}
private void setQuantity(){
   String updateStmt = "UPDATE inventory SET ";
   updateStmt = updateStmt + " quantity = " + this.theItem.getQuantity();
   updateStmt = updateStmt + " WHERE store_no = " +
     this.theItem.getStoreNumber();
   updateStmt = updateStmt + " AND item_code = " + this.theItem.getCode();
   try{
      this.theStatement = this.theConnection.createStatement();
      this.theStatement.executeUpdate(updateStmt);
      System.out.println("Organic Shop: Row has been updated.");
      this.theStatement.close();
   } catch (SQLException sqlx){
      sqlx.printStackTrace();
   }
}
```

O código abaixo é usado para invocação. Da mesma maneira, o código é escrito em Main.java. Note que o atributo quantidade de theItem foi modificado. Então, passamos o theItem para o

método updateQuantityItem() para atualização.

```
//The following code is written in Main.java

//Assume that an item has been used before.
//Let us set the new quantity.
theItem.setQuantity(1340);

//Call to update the quantity at hand
theInventory.updateQuantityItem(theItem);

theStore = theInventory.getInventoryByStore(10);
if (theStore != null){
    System.out.println(theStore.toString());
} else {
    System.out.println("The Store is null.");
}
```

3.5. Eliminando um Registro

Finalmente, não irá vender quaisquer dos itens inseridos anteriormente. O código abaixo demonstra como eliminar um registro. Note que eles tem estrutura de código similar aos processos de insersão e atualização. O método removeItem() executa a remoção. A string deleteStmt define a instrução DELETE.

```
// This code is written in DBInventory.java
public void deleteItem(Item theItem1){
   this.theItem = theItem1;
   this.establishConnection();
   this.removeItem();
   this.closeConnection();
}
private void removeItem(){
  String deleteStmt = "DELETE FROM inventory ";
  deleteStmt = deleteStmt + "WHERE store_no = " +
    this.theItem.getStoreNumber();
  deleteStmt = deleteStmt + " AND item_code = " + this.theItem.getCode();
  try{
     this.theStatement = this.theConnection.createStatement();
     this.theStatement.executeUpdate(deleteStmt);
     System.out.println("Organic Shop: Row has been deleted.");
     this.theStatement.close();
  } catch (SQLException sqlx){
     sqlx.printStackTrace();
  }
}
```

A invocação do método *deleteItem*() é mostrada no código abaixo que está escrito em Main.java. Estaremos passando o objeto theItem para ser excluído.

```
theInventory.deleteItem(theItem);
```

Todos os exemplos usam objetos da classe Statement. Existem duas outras classes que podem ser usadas que herdam de Statement.

1. A classe PreparedStatement. A principal diferença entre Statement e PreparedStatement é que PreparedStatement é pré-compilada. Quando você usa um objeto Statement para executar uma instrução SQL, a instrução é enviada direto para o DBMS, onde será compilada. Com PreparedStatement, o DBMS poderá simplesmente executar a versão já compilada da instrução SQL.

O objeto PreparedStatement também pode ser usado com instruções SQL que não recebem qualquer parâmetro. Embora, na grande maioria das vezes, você irá utilizar instruções SQL que recebem parâmetros. A vantagem de utilizar instruções SQL que recebem parâmetros é que você pode usar a mesma instrução suprindo-a com diferentes valores a cada vez que executá-la. O

exemplo abaixo mostra como usar um objeto PreparedStatement com parâmetros, trata-se de uma modificação do método setQuantity() definido na sessão Atualizando um Registro.

```
private void setQuantity(){
   String updateStmt = "UPDATE inventory SET ";
   updateStmt = updateStmt + " quantity = ?";
   updateStmt = updateStmt + " WHERE store_no = ?";
   updateStmt = updateStmt + " AND item_code = ?";
   PreparedStatement thePreparedStatement;
   try{
      this.thePreparedStatement =
        this.theConnection.createStatement(updateStmt);
      this.thePreparedStatement.setInt(1,theItem.getQuantity());
      this.thePreparedStatement.setInt(2,theItem.getStoreNumber());
      this.thePreparedStatement.setInt(3,theItem.getItemCode());
      this.thePreparedStatement.executeUpdate(updateStmt);
      System.out.println("Organic Shop: Row has been updated.");
      this.thePreparedStatement.close();
   } catch (SQLException sqlx){
      sqlx.printStackTrace();
   }
}
```

Objetos PreparedStatement são criados usando o método `createStatement` de um objeto Connection object, da mesma maneira que um objeto `Statement()` é criado. As seguintes instruções Java criam um objeto `PreparedStatement` chamado `thePreparedStatement`.

```
String updateStmt = "UPDATE inventory SET ";
updateStmt = updateStmt + " quantity = ?";
updateStmt = updateStmt + " WHERE store_no = ?";
updateStmt = updateStmt + " AND item_code = ?";
this.thePreparedStatement = this.theConnection.createStatement(updateStmt);
```

A string `updateStmt` representa a instrução UPDATE-statement que modifica o atributo quantidade de um item em particular da loja. O sinal (?) na string o lugar marcado para os parâmetros ou valores passados para a instrução SQL-statement. Os valores serão supridos com a chamada de métodos `setXXX` apropriados definidos na classe `PreparedStatement`. Se o valor que você quer substituir por um sinal (?) é um int Java, você chamará um método `setInt()`. Se o valor que você deseja subistituir é uma String Java, você deverá chamar um método `setString()`, e assim sucessivamente. Em geral, existe um método `set`*`XXX()`* para cada tipo jprimitivo da linguagem Java. No nosso exemplo, as seguintes instruções Java atribuem os valores dos parâmetros.

```
this.thePreparedStatement.setInt(1,theItem.getQuantity());
this.thePreparedStatement.setInt(2,theItem.getStoreNumber());
this.thePreparedStatement.setInt(3,theItem.getItemCode());
```

A primeira instrução Java (thePreparedStatement.setInt(1,theItem.getQuantity())) atribui o valor para o primeiro parâmetro do objeto thePreparedStatement usando o valor retornado por theItem.getQuantity(). O primeiro argumento do método chama setInt(), que é 1, representando a posição do parâmetro em PreparedStatement. O segundo argumento, é o valor da atribuição.

Da mesma forma, a segunda instrução, configura o segundo parâmetro de thePrepareStatement utilizando o valor retornado por theItem.getStoreNumber(). Por último, a terceira instrução substitui o terceiro parâmetro de thePreparedStatement pelo valor retornado por theItem.getItemCode(). Para ececutar a thePreparedStatement, o método executeUpdate() é invocado.

Será melhor usar o objeto PreparedStatement em situações que requeram mais atualizações porque, em termos de performance, ele ajuda o sistema a suprimir a fase de compilação toda vez que uma instrução SQL-statement é executada. As instruções SQL-statement são pré-compiladas.

Note que uma vez que um parâmetro é configurado com um valor, ele matém o valor até que um novo valor lhe seja atribuído, ou o método clearParameters() seja chamado.

A modificação do código para usar a classe PreparedStatement no lugar da classe Statement fica por sua conta como um exercício deste capítulo.

2. A classe CallableStatement. JDBC lhe permite chamar uma stored procedure ou stored function em uma aplicação usando um objeto CallableStatement. Assim como Statement e PreparedStatement, isto é feito com um objeto Connection aberto. Um objeto callableStatement contém apenas uma chamada para uma stored procedure ou function; ele não contém as próprias storeds procedures ou função itself.

A classe CallableStatement é uma subclasse de PreparedStatement, logo, um objeto CallableStatement pode receber parâmetros assim como um objeto PreparedStatement. Além disso, um objeto CallableStatement possui parâmetros de saída, ou parâmetros que servem às duas coisas entrada e saída. São os parâmetros INOUT (input/output). Estes parâmetros e o método execute são usados raramente. Um exemplo com o uso de CallableStatement é apresentado na sessão JDBC Stored Procedures e Functions nesta lição.

4. Servidor de Banco de Dados-do lado Programação

Aplicações-do lado métodos são métodos invocados do lado da aplicação. ***Banco de Dados-do lado das procedures*** são métodos invocados no âmbito do Banco de Dados, por exemplo, JavaDB.

Banco de Dados-do lado procedure podem ser os mesmos métodos lado-aplicação. A diferença esta como são invocados. Métodos são escritos uma única vez e onde estes são invocados-aplicações ou internas SQL-instruções- determina se uma aplicação é do lado da ou método do lado do banco de dados.

Esta seção lida com o detalhe com métodos do lado do banco de dados database-side, i.e., métodos que são invocados no banco de dados. Estas invocações definidas em stored procedures ou functions. Nesta seção discutiremos especificamente especiais pra programação para JavaDB.

4.1. JDBC Stored Procedures e Functions

Uma ***Stored Procedure*** é uma subrotina que é avaliada para acessar o sistema de banco de dados relacional. Esses representam todas operações semelhantes a enterAnOrder() ou createNewCustomer(). ***Uma Stored Function*** calcula resultados escalares, e impondo obrigatório domínio. JavaDb implementam o corpo de procedures e functions em métodos Java , i.e., chamandos de métodos Java. A habilidade para escrever functions e procedures na ponte Java completa um atributo da APIs Java no ambiente SQL na lógica do lado do servidor. Uma function ou procedure podem chamar de biblioteca padrão Java , de qualquer padrão de extensão Java , ou outras bibliotecas de terceiros que poderemos mostrar mais tarde nos exemplos.

JavaDB atualmente suporta escrever procedures em linguagem de programação Java , which segue o Padrão SQL . Com estas procedures Java, a implementação da procedure, um método estático público Java na classe Java, é compilado fora do banco de dados, tipicamente arquivo dentro de um arquivo jar e apresenta para o banco de dados com CREATE PROCEDURE-instrução ou CREATE FUNCTION-instrução. Deste modo, o CREATE PROCEDURE-instrução ou CREATE FUNCTION-instrução é conhecida como operação atômica "definir e armazenar". A compilação Java para uma procedure ou function pode armazenar no banco de dados usando o padrão SQL procedure SQLJ.INSTALL_JAR ou pode ser armazenado fora do banco de dados no path de classe da aplicação.

Diferenças entre Procedures e Functions

Estes são uma sobreposição entre stored procedure e functions mas cada uma pode também fazer coisas que a outra não pode. Ale disso, a sintaxe para invocar são diferentes. Um stored function pode ser executar uma parte da instrução SQL semelhante aquela SELECT-clausula ou a WHERE-clausula. Estas podem invocar triggers; todavia, não podem modifica dados de um banco de dados.

Para invocar uma stored procedure no Editor SQL, usamos o CALL-instrução enquanto para invocar uma function, usamos VALUES-instrução. Comparação da stored procedure e functions é listada na Tabela 4.

Descrição	Procedure	Function
Execução numa Triger	✗ (but ✓ in 10.2)	✓
Retorno do resultado efetivado(s)	✓	✗
Processo OUT/INOUT Parâmetros	✓	✗
Executar SQL SELECT	✓	✓
Executar INSERT/UPDATE/DELETE	✓	✗
Executar DDL semelhante a um CREATE e DROP	✓	✗
Executar numa SQL Expression	✗	✓

Tabela 4: Compação de Stored Procedures e Functions

Passos na Criação de Stored Procedure e Functions:

1. Criar os Métodos Java que poderão servir no corpo de stored procedure ou function. Arquivar dentro de um arquivo jar. No código abaixo, temos a definição de uma classe chamada Routines que contém dois métodos, de nomes,, insertNewItem() que insere um novo item na tabela, e countItem() que retorna o número de itens vendidos de um particular estoque. A classe é arquivada com o nome organicshoputils.jar.

Quando executamos a stored procedure, o método usado para definir o corpo da procedure mais necessárias para reusar o conexão corrente para banco de dados. Este tipo de conexão é conhecida como uma conexão aninhada. A maneira de obter uma conexão aninhada é usando a url, jdbc:default:connection que é mostrada na instrução a seguir:

```
Connection theConnection =
  DriverManager.getConnection("jdbc:default:connection","nbuser","nbuser");
```

```
package organicshop.proc;

import java.sql.*;

public class Routines {
  public static void insertNewItem(String itemName) throws SQLException {
    //Conexão Aninhada
    Connection theConnection =
      DriverManager.getConnection("jdbc:default:connection",
        "nbuser","nbuser");
    PreparedStatement theStatement = theConnection.prepareStatement(
      "INSERT INTO item VALUES (DEFAULT, ?)");
    theStatement.setString(1, itemName);
    theStatement.execute();
    theStatement.close();
    theConnection.close();
  }
  public static int countItem(int storeNumber) throws SQLException {
    int count = 0;
    //Conexão Aninhada
    Connection theConnection = DriverManager.getConnection(
      "jdbc:default:connection","nbuser","nbuser");
    PreparedStatement  theStatement = theConnection.prepareStatement(
      "SELECT COUNT(*) FROM inventory WHERE store_no = ?");
    theStatement.setInt(1, storeNumber);
    theStatement.execute();
    ResultSet theResultSet = theStatement.getResultSet();
    if (theResultSet.next()){
      count = theResultSet.getInt(1);
    }
    theResultSet.close();
    theStatement.close();
    theConnection.close();
    return count;
  }
}
```

2. Armazenar no arquivo jar no banco de dados usando o método sqlj. Na ordem para o NetBeans para trabalhar com semelhantes métodos, o derbytools.jar deve ser parte da biblioteca. Para carregar o arquivo jar para o banco de dados database, chame o método sqlj.install(). Um exemplo é mostrado abaixo:

```
CALL sqlj.install_jar
('/Users/weng/NetBeansProjects/organicshoputils/dist/organicshoputils.jar',
  'APP.OrganicShop', 0);
```

O primeiro argumento é o arquivo jar (usando o caminho absoluto), e o segundo argumento é identificar para o arquivo jar. O nome do identificador segue a convenção do padrão

SQLIdentifier92. A figura a seguir mostra como este é executado no NetBeans.

Os próximos passos para registra o classpath para incluir o arquivo jar. Este é feito através da fixação da propriedade derby.database.classpath. Usando o seguinte:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
  'derby.database.classpath', 'APP.OrganicShop');
```

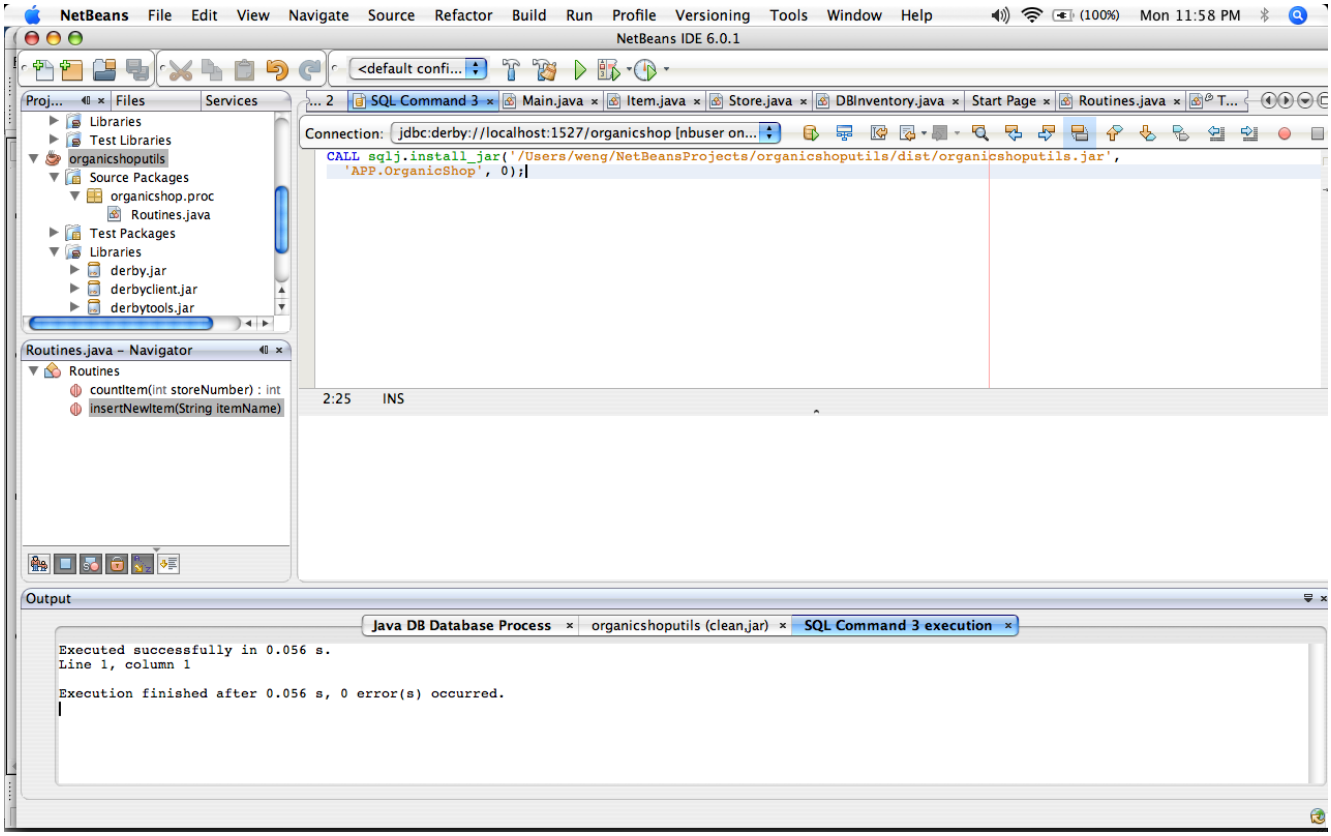

Figura 9: Executando métodos sqlj no NetBeans

3. Crie o procedimento ou função utilizando os comandos SQL *CREATE PROCEDURE* e *CREATE FUNCTION* respectivamente.

O comando *CREATE PROCEDURE* permite a criação de procedimentos armazenados Java, que podem ser chamados utilizando o comando *CALL*. A sintaxe é apresentada na Figura 9.

O nome do procedimento segue as regras do SQL92Identifier. ProcedureParameter mostra como definir parâmetros para o procedimento. *IN*, *OUT*, e *INOUT* define o parâmetro. Se não especificado, o valor padrão para um parâmetro é *IN*. O nome do parâmetro deve ser único dentro de um procedimento.

```
CREATE PROCEDURE procedure-Name ( [ ProcedureParameter
   [, ProcedureParameter] ] * ) [ ProcedureElement ] * procedure-Name

ProcedureParameter:
[ { IN | OUT | INOUT } ] [ parameter-Name ] DataType

ProcedureElement:
{
  | [ DYNAMIC ] RESULT SETS INTEGER
  | LANGUAGE { JAVA }
  | EXTERNAL NAME string
  | PARAMETER STYLE JAVA
  | { NO SQL | MODIFIES SQL DATA | CONTAINS SQL | READS SQL DATA }
}
```

Figura 10: Sintaxe do comando CREATE PROCEDURE

Tipos de dados como BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, e XML não são permitidos como parâmetros em um comando CREATE PROCEDURE.

A Tabela 5 lista os elementos do procedimento do comando CREATE PROCEDURE.

Elemento do Procedimento	Descrição
DYNAMIC RESULT SETS integer	Indica o limite superior estimado dos resultados retornados para o procedimento. O padrão é nenhum (zero) resultado dinâmico.
LANGUAGE JAVA	Indica que o gerenciador do banco de dados irá chamar o procedimento como um método estático público em uma classe Java.
EXTERNAL NAME string	Especifica o método Java a ser chamado quando o procedimento é executado, e toma a seguinte forma: nome_da_classe.nome_do_método O Nome Externo não pode ter nenhum espaço entre suas letras.
PARAMETER STYLE JAVA	Isto especifica que o procedimento utilizará uma convenção de passagem de parâmetros em conformidade com as especificações da linguagem Java e com as Rotinas SQL. Parâmetros INOUT e OUT serão passados com o matrizes simples de entrada para facilitar os valores retornados. Os resultados são retornados por parâmetros adicionais para o método Java do tipo java.sql.ResultSet [] que são passados por matrizes simples de entrada.
CONTAINS SQL	Isto indica que os comandos SQL que não lêm nem modificam dados SQL podem ser executados por procedimentos armazenados. Comandos que não são suportados por nenhum procedimento armazenado retornam um erro diferente.
NO SQL	Isto indica que o procedimento armazenado não pode executar nenhum comando SQL.
READS SQL DATA	Isto indica que alguns comandos SQL que não modificam dados podem ser incluidos em um procedimento armazenado. Comandos que não são suportados em nenhum procedimento armazenado retornam um erro diferente.
MODIFIES SQL DATA	Isto indica que um procedimento armazenado pode executar qualquer comando SQL com exceção dos comandos que não são suportados em um procedimento armazenado.

Tabela 5: Elementos do Procedimento

Os elementos de procedimento podem aparecer em qualquer ordem, mas cada tipo de elemento só pode aparecer uma vez. Uma definição de procedimento deve conter estes elementos:

- LANGUAGE
- PARAMETER STYLE
- EXTERNAL NAME

O código a seguir mostra um exemplo de CREATE PROCEDURE para o método insertNewItem() da classe Routines. O nome do procedimento no banco de dados é insertItem();

```
CREATE PROCEDURE insertItem(IN name VARCHAR(50))
PARAMETER STYLE JAVA
LANGUAGE JAVA
EXTERNAL NAME 'organicshop.proc.Routines.insertNewItem'
MODIFIES SQL DATA;
```

Figura 11: Exemplo de comando CREATE PROCEDURE

O comando CREATE FUNCTION cria funções Java, às quais você pode utilizar em uma expressão. A sintaxe é exibida no código abaixo.

```
CREATE FUNCTION function-name ( [ FunctionParameter
   [, FunctionParameter] ] * ) RETURNS ReturnDataType [ FunctionElement ] *
   function-Name
```

```
FunctionParameter:
[ parameter-Name ] DataType

FunctionElement:
{
  | LANGUAGE { JAVA }
  | EXTERNAL NAME string
  | PARAMETER STYLE ParameterStyle
  | { NO SQL | CONTAINS SQL | READS SQL DATA }
  | { RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT }
}
```

Os nomes dos parâmetros da função devem ser únicos em uma função. Tipos de dados, tais como, BLOB, CLOB, LONG VARCHAR, LONG VARCHAR FOR BIT DATA, e XML não são permitidos como parâmetros em um comando CREATE FUNCTION. A Tabela 6 mostra a definição dos elementos da função.

Elemento do Procedimento	Descrição
LANGUAGE JAVA	Indica que o gerenciador do banco de dados irá chamar o procedimento como um método estático público em uma classe Java.
EXTERNAL NAME string	Especifica o método Java a ser chamado quando o procedimento é executado, e toma a seguinte forma: nome_da_classe.nome_do_método O Nome Externo não pode ter nenhum espaço entre suas letras.
CONTAINS SQL	Isto indica que os comandos SQL que não lêm nem modificam dados SQL podem ser executados por procedimentos armazenados. Comandos que não são suportados por nenhum procedimento armazenado retornam um erro diferente.
NO SQL	Isto indica que o procedimento armazenado não pode executar nenhum comando SQL.
READS SQL DATA	Isto indica que alguns comandos SQL que não modificam dados podem ser incluidos em um procedimento armazenado. Comandos que não são suportados em nenhum procedimento armazenado retornam um erro diferente.
RETURNS NULL ON NULL INPUT	Isto especifica que a função não é invocada se qualquer dos argumentos de entrada for nulo. O resultado é o valor nulo.
CALLED ON NULL INPUT	Isto especifica que a função é invocada se qualquer ou todos os argumentos de entrada forem nulos. Esta especificação quer dizer que a função deve ser codificada para testar se há valores nulos de argumentos. A função pode retornar um valor nulo ou não-nulo. Esta é a configuração padrão.

Tabela 6: Elementos da Função

Os elementos da função podem aparecer em qualquer ordem, mas cada tipo de elemento só pode aparecer uma vez. Uma definição de função deve conter estes elementos:

- LANGUAGE
- PARAMETER STYLE
- EXTERNAL NAME

O código a seguir mostra um exemplo.

```
CREATE FUNCTION countItemInStore(storeNumber INTEGER) RETURNS INTEGER
PARAMETER STYLE JAVA
LANGUAGE JAVA
READS SQL DATA
EXTERNAL NAME 'organicshop.proc.Routines.countItem';
```

4. Opcionalmente, teste o procedimento armazenado ou função utilizando o comando CALL ou o comando VALUES respectivamente. A sintaxe do comando CALL e VALUES é exibida abaixo

```
CALL procedureName([arguments [,arguments]*]);

VALUES functionName([arguments [,arguments]*]);
```

Exemplos são encontrados abaixo. O último comando SELECT chama a função armazenada `countItemInStore()` como parte da instrução SQL.

```
-- inserts new item called Spanish Paprika in the item table
CALL insertItem('Spanish Paprika');

--returns the number of items sold for store number 30
VALUES countItemInStore(30);

--use the function in a query
SELECT store_no, count(*)
FROM inventory
GROUP BY store_no
HAVING count(*) <= countItemInStore(10);
```

5. Utilize o procedimento armazenado ou função em um programa de cliente ao utilizar um objeto `CallableStatement`. O código a seguir mostra a chamada de um procedimento armazenado em um programa de cliente Java.

```
public class TestProcedure {
  public static void main(String args[]) {
    String url = "jdbc:derby://localhost:1527/organicshop";
    String username = "nbuser";
    String password = "nbuser";
    Connection theConnection;
    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
      theConnection = DriverManager.getConnection(url,username,password);
      CallableStatement theStatement =
        theConnection.prepareCall("{ CALL insertItem(?)}");
      theStatement.setString(1,"Siling Labuyo");
      theStatement.execute();
      theConnection.close();
    } catch (ClassNotFoundException cnfx){
      System.err.println("Organic Shop: Cannot Load Driver");
      cnfx.printStackTrace();
      System.exit(1);
    } catch(SQLException sqlx){
      System.err.println(
        "Organic Shop: Unable to connection to the database");
      sqlx.printStackTrace();
      System.exit(1);
    }
  }
}
```

Neste exemplo, nós instanciamos um objeto `CallableStatement` chamado `theStatement` com um argumento que chama o procedimento armazenado `insertItem()`. O comando Java é exibido abaixo:

```
CallableStatement theStatement = theConnection.prepareCall(
  {CALL insertItem(?)});
```

O sinal de interrogação (?) encontrado na chamada do procedimento é utilizado para guardar o lugar do argumento que é especificado pelo método `setString()` do comando, como especificado a seguir:

```
theStatement.setString(1, "Siling Labuyo");
```

O método aceita dois argumentos. O primeiro argumento significa a posição do argumento na chamada do procedimento. O segundo é o valor que está sendo passado. No exemplo acima, 1 significa o primeiro argumento com o valor "Siling Labuyo". Para chamar o procedimento, nos invocamos o método execute() do objeto theStatement. Como a seguir:

```
theStatement.execute();
```

Outro exemplo é exibido no Error: Reference source not found. Aqui é chamada uma função armazenada.

```
public class TestFunction {
  public static void main(String args[]){
    String url = "jdbc:derby://localhost:1527/organicshop";
    String username = "nbuser";
    String password = "nbuser";
    Connection theConnection;
    ResultSet theResultSet;
    int count = 0;
    try {
      Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
      theConnection = DriverManager.getConnection(url,username,password);
      int storeNumber = 30;
      CallableStatement theStatement =
        theConnection.prepareCall("{VALUES countItemInStore(?)}");
      theStatement.setInt(1, storeNumber);
      theResultSet = theStatement.executeQuery();
      if (theResultSet.next()){
        count = theResultSet.getInt(1);
      } else {
        count = 0;
      }
      theConnection.close();
      System.out.println("Store no. " + storeNumber + " has " + count +
        " items.");
    } catch (ClassNotFoundException cnfx){
      System.err.println("Organic Shop: Cannot Load Driver");
      cnfx.printStackTrace();
      System.exit(1);
    } catch(SQLException sqlx){
      System.err.println(
        "Organic Shop: Unable to connection to the database");
      sqlx.printStackTrace();
      System.exit(1);
    }
  }
}
```

Neste exemplo, instanciamos um CallableStatement chamado theStatement com um argumento que chama a função armazenada countItemInStore(). O comando Java é exibido abaixo:

```
CallableStatement theStatement = theConnection.prepareCall
     ({? = CALL countItemInStore(?)});
```

O sinal de interrogação (?) encontrado na chamada da função é utilizado para guardar o lugar do valor a ser retornado pela função, e do valor do argumento que é passado para a função countItemInStore(). Para ligar o sinal à saída ou entrada, utilizamos o seguinte comando Java.

```
theStatement.registerOutParameter(1,Type.INTEGER);
theStatement.setInt(2, storeNumber);
```

O primeiro comando indica que estamos esperando um valor inteiro a ser passado ao programa realizador da chamada, e está ligado ao primeiro sinal de interrogação na chamada da função. O segundo método liga o segundo sinal de interrogação com o argumento que está sendo passado para a função. Neste caso, 2 significa o segundo sinal de interrogação enquanto storeNumber é o valor que está sendo passado para a função countItemInStore(). Para chamar a função, nós

chamamos o método executeUpdate() do theStatement. Como a seguir:

```
theStatement.executeUpdate();
```

Para recuperar o valor, é utilizado o método getInt() do theStatement.

```
count = theStatement.getInt(1);
```

Deletando um Procedimento Armazenado e Funções

O comando DROP PROCEDURE é utilizado para deletar ou remover um procedimento do banco de dados. Identifica o procedimento particular a ser removido. A sintaxe é exibida abaixo.

```
DROP PROCEDURE procedure-name
```

O comando DROP FUNCTION é utilizado para deletar ou remover uma função no banco de dados. Ele identifica a função particular a ser removida. A sintaxe é exibida abaixo.

```
DROP FUNCTION function-name
```

Exemplos são exibidos logo a seguir:

```
DROP PROCEDURE insertItem;
DROP FUNCTION countItemInStore;
```

Os comandos DROP apenas removem a definição do procedimento ou função no banco de dados. Eles não removem os arquivos jar carregados no banco de dados. Para remover os arquivos jar, é preciso utilizar métodos sqlj. O método remove_jar() é utilizado para remover o arquivo no banco de dados.

```
CALL sqlj.remove_jar('APP.OrganicShop',0)
```

O comando acima remove o organicshoputils.jar que foi carregado como o corpo da estrutura do procedimento ou função.

Outro método importante do sqlj que manipula o arquivo jar é o replace_jar que realoca o arquivo existente jar no banco de dados. Normalmente, isso é útil se foi modificado parte de nosso código. Isso permite substituir o antigo arquivo jar pelo novo arquivo. Por exemplo:

```
CALL sqlj.replace_jar
('/Users/weng/NetBeansProjects/organicshoputils/dist/organicshoputils.jar',
 'APP.OrganicShop')
```

5. Exercícios

1. O método populateStoreData() e o método populateItemData() como exibidos no comando SELECT para recuperar registros no banco de dados. Modifique o código de tal forma que o objeto PreparedStatement seja utilizado.
2. O método insertItem() como exibido no comando INSERT para inserir um registro no banco de dados. Modificar o código de tal forma que o objeto PreparedStatement seja utilizado.
3. O método removeItem() como exibido no comando INSERT para inserir um registro no banco de dados. Modificar o código de tal forma que o objeto PreparedStatement seja utilizado.
4. Utilizando o JDBC Design Pattern, implemente os códigos Java para banco de dados do problema 1 no capítulo 5 (O gerente do apartamento).
    1. Criar os métodos apropriados que manipulem os registros no banco de dados que incluam:
        - incluir um registro
        - remover um registro
        - modificar as colunas de um registro
    2. Criar os procedimentos armazenados a seguir:
        - insertRoomType(), que insere uma nova classificação de quarto
        - changeRoomStatus(), que alteram os status do quarto. O status do quarto pode ser VAGO ou OCUPADO.
    3. Criar a função armazenada:
        - retornar o tipo de quarto de um quarto em particular
        - retornar o número de quartos a partir de um tipo de quarto
5. Utilizando o JDBC Design Pattern, implemente os códigos Java para banco de dados desenvolvidos no problema 2 na lição 5 (A loja de Elétrica)
    1. Criar os métodos apropriados que manipulem os registros no banco de dados que incluam:
        - incluir um registro
        - remover um registro
        - modificar de colunas de um registro
    2. Criar os procedimentos armazenados a seguir:
        - insertClient(), que insere um registro de cliente
        - changeApplianceStatus(), que altera o status do dispositivo para FIXO.
    3. Criar a função armazenada:
        - retornar o status de um dispositivo em particular
        - retornar o número de dispositivos que está sendo consertado por um técnico
6. Utilizando o JDBC Design Pattern, implemente os códigos Java para banco de dados desenvolvidos para o e-Lagyan System
    1. Criar os métodos apropriados que manipulem os registros no banco de dados que incluam:
        - incluir um registro
        - remover um registro

- modificar as colunas de um registro

2. Criar os procedimentos armazenados a seguir:
   - insertAccount(), que insere um novo cliente no sistema
3. Criar a função armazenada:
   - retornar o balanço atual de um cliente
   - retornar o número de cartões vendidos por um cliente

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.