Módulo Ib
Interface com BancoDados
JDBC - avançado

Prof. Ismael H F Santos

Ementa

- Modulo I – JDBC-avançado
  - Versões JDBC
  - JDBC 2.0
    - Interface DataSource
    - Interface RowSet
  - JDBC 2.1
    - Movimentação do Cursor
    - Atualizações Programáticas
  - JDBC 3.0

POO-Java

Versões
JDBC

Versões do padrão JDBC

JDBC 1.0

- Além do exposto até aqui, o padrão JDBC em sua versão 1.0 define meta-dados do banco e dos resultados das consultas, através das interfaces DatabaseMetaData e ResultSetMetaData

JDBC 2.0

- Define a extensão padrão **javax.sql** que:
  - cria o conceito de um *DataSource*
  - permite o uso de *pools* de conexões
  - permite o uso de transações distribuídas
  - cria o conceito de um *RowSet*

JDBC 2.1

- Estende o padrão inicial provendo:
  - movimentação livre do *cursor* no **ResultSet**
  - atualizações programáticas, via **ResultSet**
  - atualizações em lotes (*batch updates*)
  - compatibilidade com tipos SQL3 (SQL-99)
- Extensões Padrão (javax.sql)
  - Rowset Beans
  - JNDI – Java Naming and Directory Interface
  - JTS – Java Transaction Service

JDBC 3.0

- Estende o pacote e a extensão padrão provendo:
  - conceito de *savepoints* em transações
  - configuração do *pool* de conexões
  - obtenção de valores de colunas de preenchimento automático
  - manutenção de resultados abertos após a conclusão da transação
  - e outras funcionalidades

JDBC 4.0

- Estende o .... :

POO-Java

JDBC 2.0
DataSource

Interface **DataSource**

- Representa o acesso a um SGBD
- Assim como **Driver**, estabelece conexões
- As conexões criadas pelo **DataSource** podem utilizar *pools* e/ou participar de transações distribuídas

Métodos de **DataSource**

```
public static synchronized Connection getConnection()
   throws SQLException

public static synchronized Connection getConnection( String
   user, String password) throws SQLException

public static void setLoginTimeout(int seconds)

public static void setLogWriter(PrintWriter out)
```

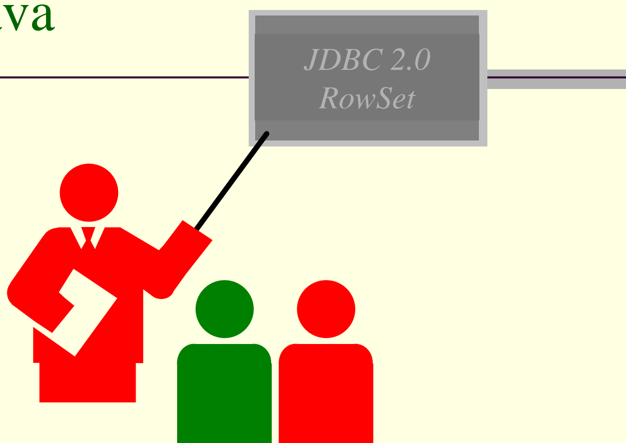
Obtendo um **DataSource**

- Um **DataSource** é obtido através do serviço de nomes e diretórios JNDI
- Uma instância do **DataSource** precisa ser criada, configurada e registrada no serviço de nomes por um administrador

Características das Conexões

- De acordo com o tipo de **DataSource**, as conexões por ele criadas poderão ser, ou não:
  - pertencentes a um *pool*
  - ser utilizadas em transações distribuídas

POO-Java



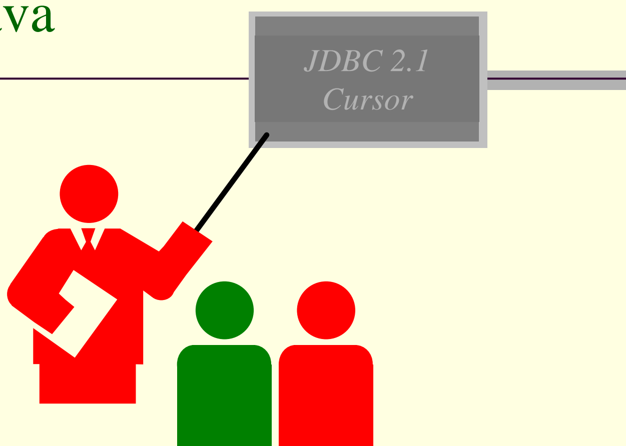
Interface **RowSet**

- É um sub-tipo **ResultSet**, seguindo o modelo *Java Beans*
- Pode ser categorizado da seguinte forma:
  - *connected rowset*
  - *disconnected rowset*
- Um **RowSet** não conectado não precisa de um **Driver** associado

Exemplos de **RowSet**

- **CachedRowSet** – um *disconnected rowset* que armazena seus dados em memória
- **JDBCRowSet** – um *connected rowset* que possibilita lidar com uma conexão como um *Java Bean*
- **WebRowSet** – um *connected rowset* que, internamente, utiliza o protocolo HTTP para se conectar a um *servlet* que provê os dados

POO-Java



Movimentação de *cursor*

- A criação de um **ResultSet** cujo *cursor* possa ser livremente movimentado deve ser feita de forma explícita, através da conexão
- A interface **Connection** possui sobrecargas dos métodos **createStatement**, **prepareStatement** e **prepareCall** para especificar as características dos **ResultSet**s correspondentes

Métodos de **Connection**

```
public Statement createStatement(
   int resultSetType, int resultSetConcurrency)
   throws SQLException

public PreparedStatement prepareStatement(
   String sql, int resultSetType, int
   resultSetConcurrency) throws SQLException

public CallableStatement prepareCall(
   String sql, int resultSetType,
   int resultSetConcurrency) throws SQLException
```

Parâmetros possíveis

- Para o tipo (**resultSetType**):
  - **TYPE_FORWARD_ONLY** (*default*)
  - **TYPE_SCROLL_INSENSITIVE**
  - **TYPE_SCROLL_SENSITIVE**
- E concorrência (**resultSetConcurrency**):
  - **CONCUR_READ_ONLY** (*default*)
  - **CONCUR_UPDATABLE**

Criando um **ResultSet** com *cursor* de livre movimentação

```
Statement stat = con.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_READ_ONLY);

ResultSet rs = stat.executeQuery(
   "SELECT * FROM Produtos");
```

Funcionou?

- O **ResultSet** criado pode não possuir as características especificadas caso o *driver* não dê suporte a essa funcionalidade

- Existem métodos na própria classe **ResultSet** para verificar:

```
public int getType() throws SQLException

public int getConcurrency() throws SQLException
```


Movimentação do *cursor*

- A movimentação do *cursor* pode ser feita da forma normal, através de **next**, ou através de outros métodos de **ResultSet**

- Esses métodos permitem mover o *cursor* para frente, para trás, para os extremos ou para uma posição arbitrária

Métodos de **ResultSet**

```
public boolean first() throws SQLException
public boolean last() throws SQLException
public boolean beforeFirst() throws SQLException
public boolean afterLast() throws SQLException

public boolean next() throws SQLException
public boolean previous() throws SQLException
public boolean relative(int rows) throws SQLException
public boolean absolute(int row) throws SQLException
```


Métodos de **ResultSet** (2)

```
public boolean isFirst() throws SQLException
public boolean isLast() throws SQLException
public boolean isBeforeFirst() throws SQLException
public boolean isAfterLast() throws SQLException

public int getRow() throws SQLException
```

Exemplos de movimentação

```
int pos = rs.getRow(); // Pega linha corrente

rs.last(); // Vai para a última

rs.absolute(10); // Vai para a décima
rs.absolute(-1); // Vai para a última
rs.absolute(-5); // Vai para antes da primeira
Rs.absolute(pos); // Vai para a linha <pos>

rs.relative(3); // Avança três linhas
rs.relative(-2); // Volta duas
```


Dicas de otimização

- Podemos dar “dicas” ao **ResultSet** para melhorar o desempenho:

```
public void setFetchSize(int rows) throws SQLException
public void setFetchDirection(int direction) throws
   SQLException

public int getFetchSize() throws SQLException
public int getFetchDirection() throws SQLException
```

POO-Java

JDBC 2.1
Atualizações
Programáticas

Atualizações programáticas

- Através do parâmetro que indica o nível de concorrência do **ResultSet**, presente nos métodos **createStatement** e outros, podemos criar **ResultSet**s onde é possível modificar programaticamente o resultado obtido (**CONCUR_UPDATABLE**)
- As operações podem ser feitas por meio de métodos específicos de **ResultSet**

Métodos de **ResultSet**

```
public void insertRow() throws SQLException
public void updateRow() throws SQLException
public void deleteRow() throws SQLException

public void updateInt(int columnIndex, int x)
   throws SQLException
public void updateString(int columnIndex, String x)
   throws SQLException
...
public void moveToInsertRow() throws SQLException
public void cancelRowUpdates() throws SQLException
```


Exemplo de atualização

```
rs.updateString("nome", "João");
rs.updateRow();

rs.moveToInsertRow();
rs.updateString("nome", "João");
rs.updateString("senha", "João");
rs.insertRow();

rs.last();
rs.removeRow();

rs.updateString("nome", "João");
rs.cancelRowUpdates();
```

Mais métodos de **ResultSet**

```
public void refreshRow() throws SQLException

public boolean rowUpdated() throws SQLException
public boolean rowInserted() throws SQLException
public boolean rowDeleted() throws SQLException

public void moveToCurrentRow() throws SQLException
```


Visibilidade de atualizações

- Atualizações realizadas na base podem ser vistas ou não por **ResultSets** que já estejam abertos, dependendo de dois fatores:
  - nível de isolamento entre transações
  - tipo do **ResultSet**
- Esse comportamento, entretanto, depende de como o SGBD é implementado

Atualização em lotes

- É possível enviar um conjunto de comandos SQL de atualização do banco de dados como um único conjunto
- Essa funcionalidade permite uma execução com melhor desempenho, em certas circunstâncias
- Basicamente, os comandos devem ser passados ao **Statement** e então executados

Métodos de **Statement**

```
public void addBatch(String sql) throws SQLException

public int[] executeBatch() throws SQLException

public void clearBatch() throws SQLException
```

Exemplo de atualização

```
try {
  con.setAutoCommit(false);
  Statement stat = con.createStatement();
  stat.addBatch("INSERT INTO xy VALUES ('0','0')");
  stat.addBatch("INSERT INTO xy VALUES ('1','1')");
  stat.addBatch("INSERT INTO xy VALUES ('2','2')");
  int[] counts = stat.executeBatch();
  con.commit();
} catch (BatchUpdateException ex) {
  int[] counts = ex.getUpdateCounts();
  ...
}
con.setAutoCommit(true);
```


Update counts

- O vetor de *update counts* retornado pelos métodos **executeBatch** e **getUpdateCount** podem ter como conteúdo:
  - um número maior igual a zero
  - **Statement.SUCCESS_NO_INFO**
  - **Statement.EXECUTE_FAILED**

PreparedStatement & CallableStatement

- Dispõem do método **addBatch()** para permitir seu uso em atualizações em lotes
- Dessa forma, podemos dar valores aos parâmetros e chamar **addBatch**, repetindo esse processo tantas vezes quanto necessário

POO-Java