# Conectividade de Banco de Dados: JDBC

## *O que é JDBC*

No verão de 1996, a Sun lançou a primeira versão do kit JDBC (Java Database Connectivity, conectividade a banco de dados Java). Esse pacote permite aos programadores se conectarem com um banco de dados, consultá-lo ou atualizá-lo, usando o SQL (Structured Query Lanquage, linguagem de consultas estruturada). (A linguagem SQL é um padrão da área de informática para acesso a banco de dados.) Achamos que esse foi um dos desenvolvimentos mais importantes na programação para a plataforma Java. Não é apenas que os banco de dados estejam entre os usos mais comuns de hardware e software atualmente. Afinal, existem muitos produtos nesse mercado; então, por que achamos que a linguagem de programação Java tem potencial para fazer um grande papel? O motivo da linguagem Java e do JDBC terem uma vantagem fundamental sobre outros ambientes de programação de banco de dados é o seguinte:

- Os programas desenvolvidos com a linguagem de programação Java e JDBC são independentes de plataforma e de fornecedor.

O mesmo programa de banco de dados, escrito em Java, pode ser executado em um computador NT, um servidor Solaris ou um aplicativo de banco de dados utilizado na plataforma Java. Você pode mover seus dados de um banco de dados para outro; por exemplo, do Microsoft SQL Server para Oracle, ou mesmo para um pequeno banco de dados incorporado em um aparelho eletrônico, e o mesmo programa ainda poderá ler seus dados. Isso contrasta fortemente com a programação de banco de dados tradicional. É também muito comum alguém escrever aplicativos de banco de dados em uma linguagem de banco de dados proprietária, usando um sistema de gerenciamento de banco de dados que esteja disponível apenas em um fornecedor. O resultado é que você pode executar o aplicativo resultante apenas em uma ou duas plataformas.

Como parte do lançamento da linguagem Java 2, em 1998, uma segunda versão da JDBC também foi publicada.

### O Projeto do JDBC

Desde o início, os desenvolvedores da tecnologia Java da Sun sabiam do potencial que a linguagem Java mostrava para se trabalhar com banco de dados. A partir de 1995, eles começaram a trabalhar na extensão da biblioteca Java padrão para lidar com o acesso via SQL aos banco de dados. O que eles esperavam fazer primeiro era estender a linguagem Java de modo que ela pudesse se comunicar com qualquer banco de dados de acesso aleatório, usando Java “puro”. Não demorou muito para que eles percebessem

que isso era uma tarefa impossível: havia simplemente bancos de dados demais no mercado, usando protocolos demais. Além disso, embora os fornecedores de banco de dados estivessem todos favoráveis à Sun no fornecimento de um protocolo de rede padrão para acesso a banco de dados, eles só estariam a favor se a Sun decidisse usar o protocolo de rede deles.

O que todos os fornecedores de banco de dados concordavam era que seria interessante se a Sun fornecesse uma API Java pura para acesso com SQL junto com um gerenciador de drivers para permitir que drivers de outros fornecedores se conectassem a banco de dados específicos. Os fabricantes de banco de dados poderiam fornecer seus próprios drivers para se ligar ao gerenciados de drivers. Haveria então um mecanismo simples para registrar drivers de terceiros junto ao gerenciador de driver – sendo o objetivo de que os drivers precisassem apenas seguir os requisitos apresentados na API do gerenciador de driver.

Após um longo período de discussão pública, a API para acesso a banco de dados se tornou a API JDBC e as regras para a escrita de drivers foram encapsuladas na API de driver JDBC. (A API de drivers JDBC é de interesse apenas para os fornecedores de banco de dados e provedores de ferramentas de banco de dados.)

Esse protocolo segue o modelo ODBC da Microsoft de muito sucesso, que forneceu uma interface da linguagem de programação C para acesso a banco de dados. O JDBC e o ODBC têm por base a mesma idéia: Os programas escritos de acordo com a API JDBC se comunicariam com o gerenciador de drivers JDBC que, por sua vez, usaria os drivers que estivessem ligados a ele nesse momento, para falar com o banco de dados real.

Mais precisamente, o JDBC consiste de duas camadas. A camada superior é a API JDBC. Essa API se comunica com a API de driver gerenciador JDBC, enviando para ela as diversas instruções SQL. O gerenciador deve (de forma transparente para o programador) se comunicar com os vários drivers de terceiros que efetivamente se conectam com o banco de dados, e retornar as informações da consulta ou executar a ação especificada por ela.

Os drivers JDBC são classificados nos seguintes tipos:

- Um driver tipo 1 transforma JDBC em ODBC e emprega com um driver ODBC para se comunicar com o banco de dados. A Sun inclui um driver assim, a ponte JDBC/ODBC (JDBC/ODBC Bridge), no JDK. Entretanto, a ponte não oferece suporte a JDBC2 e exige a distribuição e a configuração correta de um driver ODBC. A ponte é útil para testes, mas não a recomendamos para uso em produção.

- Um driver tipo 2 é um driver parcialmente escrito na linguagem de programação Java e parcialmente em código nativo, que se comunica com a API cliente de um banco de dados. Quando você usa tal driver, deve instalar algum

código específico da plataforma, além de uma biblioteca Java.

- Um driver tipo 3 é uma biblioteca cliente Java pura que usa um protocolo independente de banco de dados para comunicar pedidos do banco de dados para um componente servidor, o qual transforma os pedidos em um protocolo específico do banco de dados. A biblioteca cliente é independente do banco de dados real, simplificando assim a distribuição.

- Um driver tipo 4 é uma biblioteca Java pura que transforma pedidos JDBC diretamente em um protocolo específico do banco de dados.

A maioria dos vendedores de banco de dados fornece um driver tipo 3 ou tipo 4 com o banco de dados. Além disso, várias outras empresas são especializadas na produção de drivers com melhor atendimento aos padrões, suporte a mais plataformas ou, em alguns casos, simplemente melhor confiabilidade do que os drivers fornecidos pelos fabricantes de banco de dados.

Resumindo, o objetivo final do JDBC é tornar possível o seguinte:

- Que os programadores possam escrever aplicativos na linguagem de programação Java para acessar qualquer banco de dados, usando instruções SQL padrão ou mesmo extensões especializadas de SQL, enquanto ainda sigam as convenções da linguagem Java. (Todos os drivers JDBC devem oferecer suporte pelo memos para a versão básica do SQL 92.)

- Que os fabricantes de banco de dados e de ferramentas de banco de dados possam fornece os drivers de baixo nível. Assim, eles podem otimizar seus drivers para seus produtos específicos.

## *Conceitos Básicos de Programação JDBC*

A programação com as classes JDBC não é, conceitualmente, muito diferente da programação com as classes normais da plataforma Java: você constrói objetos a partir das classes JDBC principais, estendendo-as através de herança, se necessário.

### URLs de Banco de Dados

Ao se conectar com um banco de dados, você deve especificar a fonte dos dados e talvez precise especificar parâmetros adicionais. Por exemplo, os drivers de protocolo de rede podem precisar de uma porta e os drivers ODBC podem precisar de vários atributos.

Conforme você poderia esperar, o JDBC usa uma sintaxe semelhante ao utilizados nas URLs normais da rede para descrever origens de dados. Aqui estão exemplos da sintaxe:

```
jdbc:mysql://localhost/corejava
```

Esse comando acessaria uma origem de dados ODBC chamada COREJAVA,

usando a ponte JDBC/ODBC. A sintaxe é:

```
jdbc:nome do subprotocolo:outras informações
```

O subprotocolo é usado para selecionar o driver específico para conexão com o banco de dados.

O formato do parâmetro “outras informações” depende do subprotocolo usado. A Sun recomenda que, se você estiver usando um endereço de rede como parte do parâmetro “outras informações”, use a convenção de atribuição de nomes URL padrão //nome_host:porta/outros. Por exemplo:

```
jdbc:odbc://whitehouse.gov:5000/CATS;PWD=Hillary
```

Esta URL permitiria estabelecer uma conexão com o banco de dados CATS na porta 5000 de whitehouse.gov, usando o valor de atributo ODBC PWD configurado como “Hillary”.

## Estabelecendo a Conexão

DriverManager é a classe responsável por selecionar drivers de banco de dados e criar uma nova conexão a banco de dados. Entretanto, antes que o gerenciador de drivers possa ativar um driver, este deve ser registrado.

Existem dois métodos para registrar drivers. Seu programa pode configurar a propriedade de sistema jdbc.drivers como uma lista de drivers. Por exemplo, seu aplicativo pode incluir um arquivo de propriedade com a linha abaixo nas propriedades de sistema:

```
jdbc.drivers=com.mysql.jdbc.Driver:com.foo.aDriver
```

Essa estratégia permite aos usuários de seu aplicativo instalar os drivers apropriados simplesmente modificando um arquivo de propriedades.

A propriedade jdbc.drivers contém uma lista de nomes de classe para os drivers que o gerenciador de drivers pode usar. Os nomes são separados por dois-pontos.

Você precisa descobrir os nomes das classes de driver usadas pelo ser fornecedor, como com.pointbase.jdbc.jdbcDriver para o driver PointBase ou sum.jdbc.odbc.JdbcOdbcDriver, para a ponte JDBC/ODBC. Você também precisa inserir o código do driver em algum ligar no caminho da classe, para garantir que o aplicativo possa carregar a classe.

Alternativamente, você pode registrar um driver manualmente, carregando sua classe. Por exemplo, para carregar o driver da ponte JDBC/ODBC, você usa o comando:

```
Class.forName(“com.mysql.jdbc.Driver”);
// força o registro do driver
```

Após registrar drivers, você abre uma conexão de banco de dados semelhante ao

exemplo a seguir:

```
String url = “jdbc:mysql://localhost/corejava”;
String username = “root”;
String password = “”;
Connection con = DriverManager.getConnection(url, username,
                 password);
```

O gerenciador de drivers tentará encontrar um driver que possa usar o protocolo especificado no URL do banco de dados, fazendo uma interação através dos drivers disponíveis correntemente registrados no gerenciador de drivers.

Para nosso programa exemplo, achamos conveniente usar um arquivo de propriedades para especificar o URL, nome de usuário e senha, além do driver de banco de dados. Um arquivo de propriedades típico tem o seguinte conteúdo:

```
jdbc.drivers=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/corejava
jdbc.username=root
jdbc.password=
```

Aqui está o código para ler um arquivo de propriedades e abrir a conexão de banco de dados.

```
Properties props = new Properties();
FileInputStream in = new FileInputStream(filename);
props.load(in);
String drivers = props.getProperty(“jdbc.drivers”);
if(drivers != null)
  System.setProperty(“jdbc.drivers”, drivers);
String url = props.getProperty(“jdbc.url”);
String username = props.getProperty(“jdbc.username”);
String password = props.getProperty(“jdbc.password”);
return DriverManager.getConnection(url, username, password);
```

Note que nós simplemente não incluímos as propriedades carregadas nas propriedades de sistema:

```
System.setProperties(props); // não fazemos isso
```

Essa chamada poderia sobrescrever inadvertidamente outras propriedades de sistema. Portanto, transferimos apenas a propriedade jdbc.drivers.

O método getConnection retorna um objeto connection. Você usa o objeto connection para executar consultas e instruções de ação e efetivar ou retroceder transações.

## Executando Comandos de Ação

Para executar um comando SQL, primeiro você cria um objeto Statement. O objeto Connection que você obteve da chamada a DriverManager.getConnection pode ser usado para criar objetos Statement a partir do código:

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

Em seguida, você insere a instrução que deseja executar em uma string, por exemplo:

```
String command = “UPDATE Books “ +
                 “SET Price = Price – 5.00 ” +
                 “WHERE Title NOT LIKE '%HTML 3%'”;
```

Então, você chama o método executeUpdade da classe Statement:

```
stmt.executeUpdate(command);
```

Os comandos podem ser ações com INSERT, UPDATE e DELETE, assim como comandos de definição de dados, como CREATE TABLE e DROP TABLE. Entretanto, você não pode usar o método executeUpdade para executar consultas SELECT.

O método executeUpdade retorna uma contagem das linhas afetadas pelo comando SQL. Por exemplo, a chamada a executeUpdade no exemplo anterior retorna o número de registros de livro cujos preços foram diminuídos de 5.00.

Embora não queiramos nos aprofundar no suporte a transações, queremos mostrar a você como se faz para agrupar um conjunto de instruções para formar uma transação que pode ser efetivada (commit) quando tudo correr bem ou desfeita (rollback) como se nenhum dos comandos tivesse sido executado, se um erro tiver ocorrido em um deles.

O principal motivo para agrupar comandos em transações é a integridade do banco de dados. Por exemplo, suponha que queiramos incluir um novo livro em nosso banco de dados de livros. É importante atualizarmos simultaneamente as tabelas Books, Authors e BooksAuthors. Se a atualização fosse para incluir novos registros nas duas primeira tabelas, mas não na terceira, os livros e autores não seriam correspondidos corretamente.

Se você agrupar atualizações para uma transação, esta tem êxito em sua totalidade e pode ser efetivada ou ela falha em algum lugar intermediário. Nesse caso, você pode desfazer a transação e o banco de dados desfaz automaticamente o efeito de todas as atualizações que ocorreram desde a última transação efetivada. Além disso, é garantido que as consultas relatem apenas o estado efetivado do banco de dados.

Por definição, uma conexão de banco de dados está no modo de efetivação automática (auto-commit) e cada comando SQL é efetivado no banco de dados, assim que é executado. Uma vez efetivado o comando, você não pode desfazê-lo.

Para evitar a configuração corrente do modo de efetivação automática, chame o método getAutoCommit da classe Connection.

Você desativa o modo de efetivação automática com o comando:

```
con.setAutoCommit(false);
```

Agora, você cria um objeto statement da maneira normal:

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

Chame executeUpdade qualquer número de vezes:

```
stmt.executeUpdate(command1);
stmt.executeUpdate(command2);
stmt.executeUpdate(command3);
...
```

Quando todos os comandos tiverem sido executados, chame o método commit:

```
con.commit();
```

Entretanto, se ocorrer algum problema, chame:

```
con.rollback();
```

Dessa forma, todos os comandos até a última efetivação são desfeitos automaticamente, Normalmente, você desfaz uma transação que não ela é interrompida por uma SQLException.

## Consultando com JDBC

Para fazer uma consulta, primeiro você cria um objeto Statement, conforme descrito anteriormente. Após você pode executar uma consulta simples usando o objeto executeQuery da classe Statement e fornecendo o comando SQL como uma string para a consulta. Note que você pode usar o mesmo objeto Statement para várias consultas não relacionadas.

É claro que você está interessado no resultado da consulta. O objeto executeQuery retorna um objeto do tipo ResultSet, que você usa para percorrer o resultado, um registro (linha) por vez.

```
ResultSet rs = stmt.executeQuery(“SELECT * FROM Books”);
```

O laço básico para analisar um conjunto de resultado, é semelhante ao seguinte:

```
while(rs.next()) {
  examina uma linha do conjunto de resultado
}
```

Ao inspecionar uma linha específica, você desejará conhecer o conteúdo de cada coluna. Um grande número de métodos “get” fornece essa informação.

```
String isbn = rs.getString(1);
float price = rs.getDouble(“Price”);
```

Existem um método para cada tipo da linguagem de programação Java, como

getString e getDouble. Cada método “get” tem duas formas, uma que recebe um argumento numérico e outra que recebe um argumento string. Quando fornece um argumento numérico, você se refere à coluna com esse número. Por exemplo, rs.getString(1) retorna o valor da primeira coluna da linha corrente.

Quando você fornece um argumento string, se refere à coluna no conjunto de resultado com esse nome. Por exemplo, rs.getDouble(“Price”) retorna o valor da coluna de nome Price. Usar o argumento numérico é muito pouco eficiente, mas o argumento string torna o código mais fácil de ler e manter.

Cada método “get” fará conversões de tipo razoáveis, quando o tipo do método não coincidir com o tipo da coluna. Por exemplo, a chamada rs.getString(“Price”); converte o valor em ponto flutuante da coluna Price em uma string.

| *Tipo de dados SQL* | *Tipo de dados Java* |
|---|---|
| INTEGER ou INT | int |
| SMALLINT | short |
| NUMERIC(m, n), DECIMAL(m, n) ou DEC(m, n) | java.sql.Numeric |
| FLOAT(n) | double |
| REAL | float |
| DOUBLE | double |
| CHARACTER(n) ou CHAR(n) | String |
| VARCHAR(n) | String |
| BOOLEAN | boolean |
| DATE | java.sql.Date |
| TIME | java.sql.Time |
| TIMESTAMP | java.sql.Timestamp |
| BLOB | java.sql.Blob |
| CLOB | java.sql.Clob |
| ARRAY | java.sql.Array |

*Tipos de dados SQL e seus tipos de dados Java correspondentes*

## Tipos SQL Avançados (JDBC2)

Além de números, strings e datas, muitos bancos de dados podem armazenar objetos grandes, como imagens ou outros dados. No SQL, os objetos binários grandes são chamados BLOBs e os objetos de caractere grandes são chamados de CLOBs. Os métodos getBlob e getClob retornam objetos de tipo Blob e Clob. Essas classes têm métodos para recuperar os bytes ou caracteres nos objetos grandes.

Um Array SQL é uma seqüência de valores. Por exemplo, em uma tabela Student, você pode ter uma coluna Scores que seja um ARRAY OF INTEGER. O método getArray retorna um objeto de tipo java.sql.Array. A interface java.sql.Array possui métodos para buscar os valores do array.

Quando você obtém um blob ou um array de um banco de dados, o conteúdo em si é recuperado do banco de dados apenas quando os valores individuais são solicitados. Esse é um aprimoramento de desempenho útil, pois os dados podem ser bastante volumosos.

## *Populando um Banco de Dados*

Nosso primeiro programa JDBC lê dados de um arquivo de texto em um formato como o seguinte:

```
Publisher_Id char(5), Name char (30), URL char(80)
'01262', 'Academic Press', 'www.apnet.com'
'18835', 'Coriolis', 'www.coriolis.com/'
```

A primeira linha do arquivo de texto lista os nomes e tipos das colunas. As linhas seguintes listam os dados a serem inseridos, em formato delimitado por vírgulas.

O programa MakeBD lê a primeira linha e a transforma em uma instrução CREATE TABLE, como segue:

```
CREATE TABLE Publishers (Publisher_Id char(5), Name char (30),
                  URL char(80))
```

Todas as outras linhas de entrada se tornam instruções INSERT, como segue:

```
INSERT INTO Publishers VALUES ('01262', 'Academic Press',
                  'www.apnet.com')
```

Você deve executar esse programa como segue:

```
java MakeDB Books
java MakeDB Authors
java MakeDB Publishers
java MakeDB BooksAuthors
```

O conteúdo do arquivo MakeDB.properties é como segue:

```
jdbc.drivers=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost/corejava
jdbc.username=root
jdbc.password=
```

Os passos a seguir fornecem uma visão geral do programa MakeDB.

1. Conecte-se com o banco de dados. O método getConnection lê as

propriedades no arquivo MakeDB.properties e inclui a propriedade jdbc.drivers nas propriedades de sistema. O gerenciados de drivers usa a propriedade jdbc.drivers para carregar as propriedades jdbc.url, jdbc.username e jdbc.password, para abrir a conexão de banco de dados.

2. Obtém o nome de arquivo a partir do nome de tabela, anexando a extensão dat (isto é, os dados da tabela Books são armazenados no arquivo Books.dat).

3. Lê os nomes e os tipos de coluna e constrói um comando CREATE TABLE. Em seguida, executa este comando:

```
String line = in.readLine();
String command = “CREATE TABLE “ +
tableName + “ (“ + line + “)”;
stmt.executeUpdate(command);
```

Aqui, usamos executeUpdate e não executeQuery, pois essa instrução não produz resultados.

4. Para cada linha no arquivo de entrada, executa uma instrução INSERT:

```
comand = “INSERT INTO “+ tableName + “
VALUES (“ + line + “)”;
stmt.executeUpdate(command);
```

5. Depois que todos os elementos tiverem sido inseridos, executa uma consulta SELECT * FROM tableName, usando o método showTable para mostrar o resultado. Esse método mostra se os dados foram inseridos com êxito. Para descobrir o número de colunas no conjunto de resultados, precisamos do método getColumnCount da classe ResultSetMetaData.

Vide o código MakeDB.java.

## Atualizações em Lote (JDBC2)

No JDBC2, você pode melhorar o desempenho do programa do exemplo anterior, usando uma atualização em lote (batch). Em uma atualização em lote, uma seqüência de comandos é reunida e enviada como um lote.

Os comandos em um lote podem ser ações como INSERT, UPDATE e DELETE, assim como comandos de definição de dados, como CREATE TABLE e DROP TABLE. Entretanto, você não pode incluir comandos SELECT em um lote, pois a execução de um comando SELECT retorna um conjunto de resultados.

Para executar um lote, primeiro você cria um novo objeto Statement:

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

Agora, em vez de chamar executeUpdate, você chama o método addBatch:

```
String line = in.readLine();
```

```
String command = “CREATE TABLE “ + tableName + “ (“ + line +
                 “)”;
stmt.addBatch(command);

while((line = in.readLine()) != null) {
  command = “INSERT INTO “ + tableName + “ VALUES (“ + line +
                 “)”;
  stmt.addBatch(command);
}
```

Finalmente, você envia o lote inteiro.

```
int[] counts = stmt.executeBatch();
```

A chamada a executeBatch retorna um array de contagens de linhas para todos os comandos enviados. (Lembre-se que uma chamada individual a executeUpdade retorna um inteiro, a saber, a contagem de linhas afetadas pelo comando.) Em nosso exemplo, o médoto executeBatch retorna um array com o primeiro elemento igual a 0 (pois o comando CREATE TABLE produz uma contagem de fila igual a 0) e todos os outros elementos iguais a 1 (pois cada comando INSERT afeta uma linha).

Para um tratamento de erros correto no modo de lote deve-se tratar a execução do lote como uma única transação. Se ela falhar no meio de sua execução, deve-se retroceder para o estado existente antes do inicio do lote.

Primeiro, desative o modo de efeticação automática e em seguida, reúna o lote, execute-o, efetive-o e finalmente, restaure o modo efetivação automática original:

```
boolean autoCommit = con.getAutoCommit();
con.setAutoCommit(false);
Statement stmt = con.getStatement();
...
// continua chamando stmt.addBatch(...);
...
stmt.executeBatch();
con.commit();
con.setAutoCommit(autoCommit);
```

## *Executando Consultas*

Agora, escreveremos um programa que executará consultas no banco de dados COREJAVA. Para que esse programa funcione, você deve ter preenchido o banco de dados COREJAVA com tabelas, conforme descrito anteriormente.

Neste programa, usaremos um novo recurso, instruções preparadas. Considere a

consulta de todos os livros de uma editora em particular, independente do autor. A consulta SQL é:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publishers_Id = Publishers.Publisher_Id
AND Publishers.Name = o nome da caixa de lista
```

Em vez de construir um comando de consulta sepadado sempre que o usuário ativa tal consulta, podemos preparar uma consulta com uma variável hospedeira e usá-la muitas vezes, cada vez fornecendo uma string diferente para a variável. Essa técnica nos fornece uma vantagem de desempenho. Quando o banco de dados executa uma consulta, primeiro ele calcula uma estratégia sobre como executá-la eficientemente. Preparando a consulta e reutilizando-a, você garante que a etapa de planejamento seja feita apenas uma vez. (O motivo de você não querer sempre preparar uma consulta á que a estratégia perfeita pode mudar quando seu banco de dados mudar. Você precisa ponderar o dispêndio da otimização em relação  ao dispêndio de consultar seus dados com menos eficiência.)

Cada variável “hospedeiro” em uma consulta preparada é indicada com uma “?”. Se houver mais de uma variável, então você deverá controlar as posições da “?”, ao configurar os valores. Por exemplo, nossa consulta preparada se torna:

```
String publisherQuery =
                  “SELECT Books.Price, Books.Title “ +
                  “FROM Books, Publishers “ +
                  “WHERE Books.Publishers_Id =
                  Publishers.Publisher_Id “ +
                  “AND Publishers.Name = ?”;
PreparedStatement publisherQueryStmt =
                  com.prepareStatement(publisherQuery);
```

Antes de executar a instrução preparada, você deve ligar as variáveis hospedeiras aos valores reais, com um método set. Assim como nos métodos ResultSet get, existem diferentes métodos set para os diversos tipos. Aqui, queremos configurar uma string como um nome de editora.

```
PublisherQueryStmt.setString(1, publisher);
```

O primeiro argumento é a variável hospedeira que queremos configurar. A posição 1 denota o primeiro “?”. O segundo argumento é o valor que queremos atribuir à variável hospedeira.

Se você reutilizar uma consulta preparada que já executou e a consulta tiver mais de uma variável hospedeira, todas as variáveis hospedeiras permanecerão ligadas à medida que você as configura, a não ser que as altere com um método set. Isso significa que você só precisa chamar set nas variáveis hospedeiras que mudam de uma consulta para

outra.

Uma vez que todas as variáveis tiverem sido ligadas a valores, você pode executar a consulta:

```
ResultSet rs = publisherQueryStmt.executeQuery();
```

Você processa o conjunto de resultado normalmente. Aqui, incluímos as informações na área de texto result.

```
Result.selText("");
while(rs.next())
  result.appendText(rs.gatString(1) + " | " + rs.getString(2) +
                "\n");
rs.close();
```

Existirá um total de quatro consultas preparadas nesse programa, uma para cada caso mostrado na tabela abaixo.

| ***Author*** | ***Publisher*** |
|---|---|
| qualquer um | qualquer um |
| qualquer um | especificado |
| especificado | qualquer um |
| especificado | especificado |

*Consultas selecionadas*

O recurso de atualização de preços será implementado como uma instrução UPDATE simples. Para variar, não optamos neste caso por fazer uma instrução separada. Assim chamaremos executeUpdade e não executeQuery, pois a instrução UPDATE não retorna um conjunto de resultados e não precisamos de um. O valor de retorno de executeUpdade é a contagem de linhas alteradas. Apresentaremos a contagem na área de texto.

```
String updadeStatement = "UPDATE books ...";
int r = stmt.executeUpdate(updateStatement);
result.setText(r + " records updated");
```

Os passos a seguir fornecem uma visão geral do programa.

1. Organiza os componentes no quadro, usando um layout de grade de conteúdo.

2. Preenche as caixas de texto referentes ao autor e à editora, executando duas consultas que retornam todos os nomes de autor e editora do banco de dados.

3. Quando o usuário seleciona "Query", descobre qual dos quatro tipos de consulta precisa ser executada. Se essa for a primeira vez que esse tipo de

consulta é executada, assim a variável de instrução preparada será null e a instrução preparada será construída. Em seguida, os valores são ligados à consulta e esta á executada.

As consultas envolvendo autores são mais complexas. Como um livro pode ter vários autores, a tabela BooksAuthor fornece a correspondência entre autores e livros. Por exemplo, o livro com o número de ISBN 1-56-604288-7 tem dois autores, com códigos HARR e KIDD. A tabela BooksAuthors tem as linhas descritas baixo para indicar esse fato:

```
1-56-604288-7 | HARR | 1
1-56-604288-7 | KIDD | 2
```

A terceira coluna lista a ordem dos autores. (Não podemos usar apenas a posição dos registros na tabela. Não existe um ordenamento de linhas fixo em uma tabela relacional.) Assim, a consulta precisa juntar (join) a tabela Books à tabela BooksAuthors e em seguida, à tabela Authors, para comparar o nome do autor com aquele selecionado pelo usuário.

```
SELECT Books.Price, Books.Title
FROM Books, Publishers, BooksAuthors,
Authors
WHERE Books.Publisher_Id =
Publisher.Publisher_Id
AND Books.ISBN = BooksAuthors.ISBN
AND BooksAuthors.Author = Authors.Author
AND Authors.Name = ?
```

4. Os resultados da consulta serão apresentados na caixa de texto de resultado.

5. Quando o usuário seleciona “Change price”, a consulta de atualização é construída e executada. A consulta é bastante complexa, pois a cláusula WHERE da instrução UPDATE precisa do código da editora e conhecemos apenas o nome dela. Esse problema á resolvido com uma consulta aninhada.

```
UPDATE Books
SET Price = Price + price change
WHERE Books.Publisher_Id =
   (SELECT Publisher_Id
   FROM Publishers
   WHERE Name = publisher name)
```

6. Inicializaremos a conexão e os objetos instrução na construtora. Preservaremos neles durante toda a vida do programa. Imediatamente antes que o programa termine, chamaremos o método dispose e esses objetos serão fechados.

```
class QueryDB extends Frame {
```

```
    Connection con;
    Statement stmt;

    QueryDB() {
       con =
       DriverManager.getConnection(url,user,
       password);
       ...
    }
    ...
    void dispose() {
       stmt.close();
       con.close();
    }
    ...
}
```

Vide o código QueryDB.java.

## *Conjunto de Resultados Roláveis e Atualizáveis*

Os aprimoramentos mais úteis no JDBC2 estão na classe ResultSet. Conforme você já viu, o método next da classe ResultSet faz a interação sobre as linhas de um conjunto de resultados. Normalmente, você quer que o usuário possa mover-se para frente e para trás no conjunto de resultado. Mas o JDBC não tinha um método previous. Os programadores que quisessem implementar interação inversa tinha de colocar os dados do conjunto de resultado em chache, manualmente. Os conjunto de resultados roláveis do JDBC2 permitem que você se mova para frente e para trás em um conjunto de resultados e pule para qualquer posição do conjunto.

Além disso, quando você apresenta o conteúdo de um conjunto de resultados para os usuários, eles podem ficar tentados a editá-lo. Se você fornecer um modo que possa ser editado para seus usuários, terá de certificar-se de que as edições dos usuários sejam enviadas de volta para o banco de dados. No JDBC, você tinha de programar instruções UPDATE. No JDBC2, você pode simplesmente atualizar as entradas do conjunto de resultado e o banco de dados á atualizado automaticamente.

### Conjuntos de Resultados Roláveis (JDBC2)

Para obter conjunto de resultados roláveis a partir de suas consults, você deve obter um objeto Statement diferente, com o método:

```
Statement stmt = con.createStatement(tipo, concorrencia);
```

Para uma instrução preparada, use a chamada:

```
PreparedStatement stmt = con.prepareStatement(comando, tipo,
                concorrencia);
```

Os valores possíveis de tipo e concorrência estão relacionados na Tabela “Valores de tipo de conjunto de resultados” e “Valores de concorrência de conjuntos de resultados”. Você tem as seguintes escolhas:

- Você quer que o conjunto de resultado seja rolável ou não? Se não, use ResultSet.TYPE_FORWARD_ONLY.
- Se o conjunto de resultados pode ser rolado, você quer que ele possa refletir as alterações no banco de dados que ocorreram depois da consulta que produziu? (Em nossa discussão, vamos supor a configuração ResultSet.TYPE_SCROLL_INSENSITIVE para conjuntos de resultados roláveis. Isso pressupões que o conjunto de resultados não “percebe” alterações do banco de dados que ocorreram após a consulta.)
- Você quer atualizar o banco de dados editando o conjunto de resultados?

Por exemplo, se você quer simplesmente rolar por um conjunto de resultado, mas não quer quer editar seus dados, então use:

```
Statement stmt =
                con.createStatement(ResultSet.TYPE_SCROOL_INSEN
                SITIVE, ResultSet.CONCUR_READ_ONLY);
```

| | |
|---|---|
| TYPE_FORWARD_ONLY | O conjunto de resultados não é rolável |
| TYPE_SCROOL_INSENSITIVE | O conjunto de resultados á rolável, mas não é sensível às alterações do banco de dados. |
| TYPE_SCROOL_SENSITIVE | O conjunto de resultados á rolável e é sensível às alterações do banco de dados. |

*Valores de tipo de conjuntos de resultados*

| | |
|---|---|
| CONCUR_READ_ONLY | O conjunto de resultados não pode ser usado para atualizar o banco de dados. |
| CONCUR_UPDATABLE | O conjunto de resultado pode ser usado para atualizar o banco de dados. |

*Valores de concorrência de conjuntos de resultados*

Todos os conjuntos de resultados que são retornados por chamadas do método abaixo, são roláveis.

```
ResultSet rs = stmt.executeQuery(consulta);
```

Um conjunto de resultados rolável tem um cursor que indica a posição corrente.

NOTA: Na verdade, um driver de banco de dados pode não ser capaz de atender seu pedido de cursor rolável ou atualizável. (Os métodos supportsResultSetType e supportsResultSetConcurrency da classe DatabaseMetaData informam quais tipos e modos de concorrência são aceitos por um banco de dados em particular.) Mas, mesmo que um banco de dados ofereça suporte a todos os modos de conjunto de resultados, uma consulta em particular poderia não produzir um conjunto de resultados com todas as propriedades solicitadas. (Por exemplo, o conjunto de resultados de uma consulta complexa talvez não possa ser atualizado.) Nesse caso, o método executeQuery retorna um ResultSet com menos capacidades e inclui um aviso (warning) no objeto conexão. Você pode recuperar avisos com o método getWarnings da classe Connection. Alternativamente, você pode usar os métodos getType e getCurrency da classe ResultSet, para descobrir qual modo um conjunto de resultados realmente possui. Se você não verificar os resursos do conjunto de resultados e executar uma operação são suportada, como previous, em um conjunto de resultados não rolável, será lançada uma SQLException.

A rolagem é muito simples. Você usa o código abaixo para rolar para trás.

```
if(rs.previous()) ...
```

O método retorna true, se o cursor estiver posicionado sobre uma linha real; false, se estiver posicionado antes da primeira linha.

Você pode mover o cursor para trás ou para frente por um número de linhas, com o código:

```
rs.relative(n);
```

Se n for positivo, o cursor se move para frente. Se n for negativo, ele se move para trás. Se n for zero, a chamada não tem efeito. Se você tentar mover o cursor fora do conjunto corrente de linhas, ele será configurado de forma a apontar após a última linha ou antes da primeira, dependendo do sinal de n. Desta forma, o método retorna false e o cursor não se move. O método retorna true, se o cursor estiver sobre uma linha real.

Alternativamente, você pode configurar o cursor para um número de linhas específico:

```
rs.absoluta(n);
```

Você obtém o número da linha corrente, com a chamada

```
int n = rs.getRow();
```

A primeira linha do conjunto de resultados tem o número 1. Se o valor de retorno for 0, o cursor não está atualmente em uma linha ou ele está antes da primeira ou após a última linha.

Existem métodos de conveniência para mover para a primeira, para a última, para antes da primeira ou para depois da última posição.

```
first
last
beforeFirst
afterLast
```

Finalmente, os métodos abaixo testam se o cursor está em uma dessas posições especiais.

```
isFirst
isLast
isBeforeFirst
isAfterLast
```

Usar um cursor rolável é muito simples. O trabalho árduo de colocar os dados da consulta em cache é realizado nos bastidores pelo driver do banco de dados.

## Conjuntos de Resultados Atualizáveis (JDBC2)

Se você quiser editar os dados do conjunto de resultados e ter as alterações automaticamente refletidas no banco de dados, será necessário criar um conjunto de resultados atualizável. Os conjunto de resultados atualizáveis não precisam ser roláveis, mas se você apresentar dados para um usuário para edição, normalmente vai querer permitir também a rolagem.

Para obter conjuntos de resultados atualizáveis, você cria uma instrução como a seguinte:

```
Statement stmt = con.createStatement(
                  ResultSet.TYPE_SCROLL_INSENSITIVE,
                  ResultSet.CONCUR_UPDATABLE);
```

Assim, os conjuntos de resultados retornados por uma chamada a executeQuery serão atualizáveis.

Por exemplo, suponha que você queira aumentar os preços de alguns livros, mas não tenha um critério simples para executar um comando UPDATE. Você pode fazer a iteração por todos os livros e atualizar os preços com base em condições arbitrárias.

```
String query = “SELECT * FROM Books”;
ResultSet rs = stmt.executeQuery(query);
while(rs.next()) {
  if(...) {
    double increase = ...
    double price = rs.getDouble(“Price”);
    rs.updateDouble(“Price”, price + increase);
    rs.updateRow();
  }
}
```

Existem métodos updateXxx para todos os tipos de dados que correspondam aos tipos SQL, como updateDouble, updateString etc. Assim como nos métodos getXxx, você especifica o nome ou o número da coluna e especifica o novo valor para o campo.

O método updateXxx altera apenas os valores na linha em si, não o banco de dados. Quando você tiver terminado com as atualizações de campo em uma linha, deve chamar o método updateRow. Esse método envia todas as atualizações da linha corrente para o banco de dados. Se você mover o cursor para outra linha, sem chamar updateRow, todas as atualizações serão descartadas do conjunto de linhas e elas nunca serão comunicadas para o banco de dados. Você também pode chamar o método cancelRowUpdades para cancelar as atualizações na linha corrente.

O exemplo anterior mostra como você modifica uma linha existente. Se você quiser incluir uma nova linha no banco de dados, use primeiro o método moveToInsertRow, para mover o cursor para uma posição especial, chamada linha de inserção. Você constrói uma nova linha na posição da fila de inserção, executando instruções updateXxx. Finalmente, quando você tiver terminado, chame o método insertRow para inserir a nova linha no banco de dados. Quando você tiver terminado de inserir, chame moveToCurrentRow para mover o cursor de volta para a posição anterior à chamada de moveToInsertRow. Aqui está um exemplo.

```
rs.moveToInsertRow();
rs.updateString(“Title”, title);
rs.updateString(“ISBN”, isbn);
rs.updateString(“Publisher_Id”, pubid);
rs.updateString(“URL”, url);
rs.updateDouble(“Price”, price);
rs.insertRow();
rs.moveToCurrentRow();
```

Note que você não tem influência sobre onde os dados são inseridos no conjunto de resultados ou no banco de dados.

Finalmente, você pode excluir a linha que está sob o cursor.

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

O método deleteRow remove imediatamente a linha do conjunto de resultados e do bando de dados.

Os métodos updateRow, insertRow e deleteRow da classe ResultSet proporcionam a você o mesmo poder que a execução dos comandos SQL UPDATE, INSERT e DELETE. Entretanto, os programadores que estão acostumados com a linguagem de programação Java acharão mais natural manipular o conteúdo do banco de dados através de conjuntos de resultados do que construindo instruções SQL.