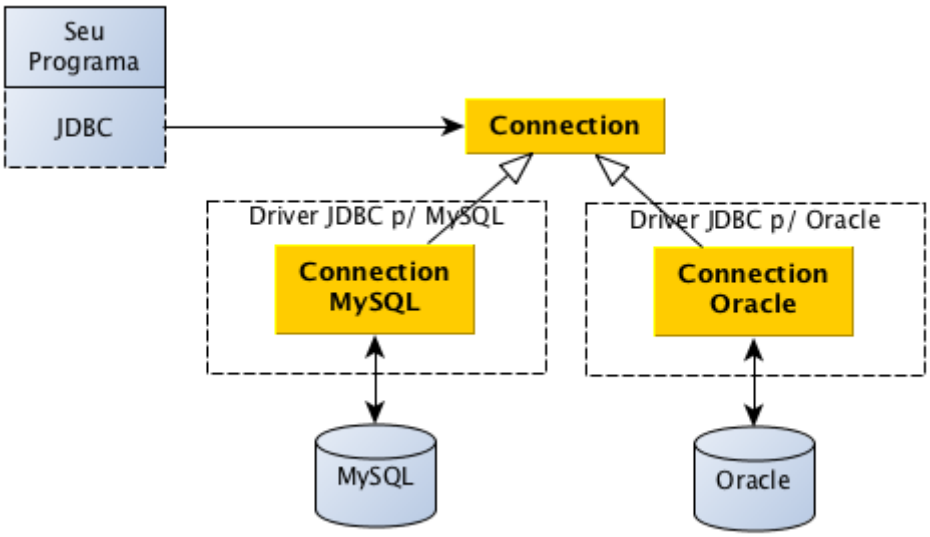
JDBC: Acessando Bancos de Dados em Java

A API JDBC

Quando queremos acessar um banco de dados, é necessário se conectar e, depois, se comunicar com ele. Essa comunicação é feita por meio de um protocolo específico para cada produto e/ou fornecedor. Assim sendo, é muito comum termos alguma biblioteca de funções ou classes para fazer esse trabalho pesado. Normalmente, o próprio fabricante do banco de dados fornece essa biblioteca sob o nome de “driver de acesso”. E, seu programa “conversa” diretamente com esse “driver”. Isso significa que, se você precisar trocar o acesso do seu programa para outra “marca” de banco de dados, precisará de outro “driver” e, provavelmente, terá que alterar seu código porque a utilização dessa nova biblioteca será diferente. Imagina a dor de cabeça…

O Java possui uma API (Application Programming Interface) para acesso a banco de dados chamada JDBC (Java DataBase Connectivity). Essa API permite usar os “drivers de acesso” de forma transparente, isto é, independentemente da “marca” do banco de dados. Com isso, se sua aplicação fizer uso apenas de comandos SQL básicos, é bem provável que ela fique extremamente flexível em relação ao banco de dados que você for usar.

Mas, como isso é possível? Bom, isso é possível pelo bom uso dos conceitos de Programação Orientada e Objetos e mais alguns truquezinhos “avançados” (como Reflection, Service Providers, entre outros). No fim, a “mágica” toda fica por conta do polimorfismo. Vejamos como, observando a figura abaixo:



Note que o seu programa continua usando um driver, mas a “conversa” é feita indiretamente, por meio do JDBC e sua interface chamada Connection. Cada “Driver JDBC” implementa essa interface de forma específica para cada banco de dados. Usando os conceitos de Orientação a Objetos, seu programa “sabe” apenas da interface Connection, ficando livre de acessar classes e/ou funções específicas, ou seja, o código fica muito flexível. Observação: obviamente, o JDBC é composto por “muuuitas” outras interfaces e classes… Connection é apenas uma delas!

Mas, como é instanciado o objeto correto? Bom, o JDBC possui um classe chamada DriverManager que sabe ler uma configuração chamada “URL de conexão” (ou “string de conexão”) e, por meio dela, cria o objeto específico para o banco que se quer usar. Isso ficará

mais claro logo abaixo, quando você vir o código!

A Receita de Bolo

Para começar, saiba que precisamos, basicamente, de quatro informações para acessar um banco de dados:

Nome completo (FQN) da classe que implementa o Driver JDBC.

URL de conexão.

Nome do usuário de acesso ao banco.

Senha do usuário de acesso ao banco.

Para saber os dois primeiros itens, é necessário consultar o manual do Driver JDBC específico para o banco que você quer acessar. Ah! Lógico, você também irá precisar fazer o download do driver e colocá-lo junto ao seu programa (cada IDE faz de um jeito, mas, basicamente, a JVM irá acessa-lo por meio do CLASSPATH).

Por exemplo, se formos acessar um banco de dados MySQL, precisaremos fazer o download do Driver JDBC para MySQL e, olhando a documentação, saberemos que o nome completo da classe que implementa o Driver JDBC é “com.mysql.jdbc.Driver” e que a URL de conexão tem o seguinte formato: “jdbc:mysql://host:port/database“. Observação: para os exemplos deste artigo, estarei usando um banco de dados de exemplo do MySQL chamado “sakila“.

Uma vez que temos todas as informações, carregamos o driver e nos conectamos ao banco usando as seguintes linhas de código:

```
1    // informações necessárias
2    String driver = "com.mysql.jdbc.Driver";
3    String url = "jdbc:mysql://localhost:3306/sakila";
4    String user = "root";
5    String pass = "god";
6
7    // carrega o driver
8    Class.forName(driver);
9
10   // faz a conexão
11   Connection conn = DriverManager.getConnection(url, user, pass);
```

Consultas

Após obtida uma conexão, podemos enviar consultas ao banco de dados por meio de comandos SQL. Para isso, precisamos primeiro preparar a consulta e, depois, executá-la. Após isso, você obterá um conjunto de resultados pelo qual deverá iterar, verificando se tem um

próximo resultado. Caso tenha, use-o como quiser! Vamos ver o código devidamente comentado?

```
1    // sua consulta
2    String sql = "select * from customer order by first_name, last_name";
3
4    // prepara a consulta, por meio da conexão (conn)
5    PreparedStatement stmt = conn.prepareStatement(sql);
6
7    // executa a consulta, obtendo um conjunto de resultados (ResultSet)
8    ResultSet rs = stmt.executeQuery();
9
10   // itera pelo conjunto de resultados, perguntando se tem um próximo (next)
11   while (rs.next()) {
12       // se tem, obtém os campos que você quiser
```

```
13      String nome = rs.getString("first_name");
14      String sobrenome = rs.getString("last_name");
15      int ativo = rs.getInt("active");
16      int loja = rs.getInt("store_id");
17
18      // e, usa os valores aqui... exemplo: mostra na tela:
19      System.out.println(loja + " - " + nome + " " + sobrenome + " - " + ativo);
20  }
```

O interessante é que, ao se preparar a consulta, podemos colocar parâmetros nela. Como assim? Imagine que você quer consultar os “customers” que não estão ativos (active = 0) e que são da “store” 2 (store_id = 2). Podemos criar a seguinte consulta:

```
1 String sql = "select * from customer where active = 0 and store_id = 2 order by first_name, last_name";
```

No entanto, pode ser que esses valores (active = 0 e store_id = 2) possam ser passados pelo usuário, virem de uma variável, etc. Assim, seria interessante se deixássemos eles “em aberto”, por assim dizer. Fazemos isso dessa forma:

```
1 String sql = "select * from customer where active = ? and store_id = ? order by first_name, last_name";
```

Note que coloquei dois símbolos de interrogação “?“. O primeiro (1) deve ser substituído pelo valor 0 e o segundo (2) pelo valor 2. Para substituir, usamos o seguinte código:

```
1 // sua consulta
2 String sql = "select * from customer where active = ? and store_id = ? order by first_name, last_name";
3
4 // prepara a consulta, por meio da conexão (conn)
5 PreparedStatement stmt = conn.prepareStatement(sql);
6
7 // configura os parâmetros da consulta
8 stmt.setInt(1, 0); // o primeiro deve ser substituído pelo valor 0
9 stmt.setInt(2, 2); // o segundo deve ser substituído pelo valor 2
```

Veja que usamos os métodos setInt, setString, etc. para substituir os parâmetros por valores, da mesma forma que usamos os métodos getInt, getString, etc. para obter os valores dos campos no conjunto de resultados.

Atualizações

Nesse ponto, já sabemos como consultar o banco de dados. Mas, e se quisermos alterar (update) alguma coisa por lá? “Tipo”, se inserirmos, atualizarmos ou removermos dados, estaremos alterando (updating) o estado do nosso banco. Faz sentido? Bom, para fazer isso, o código é “beeem” parecido com o anterior. A mudança fica por conta do método que temos que chamar. Para executar a consulta (query), chamamos executeQuery. Para executarmos uma alteração (update), chamamos executeUpdate. Exemplo: se quisermos remover todos os “customers” que não estão ativos, escreveríamos o seguinte código:

```
1    // sua consulta
2    String sql = "delete from customer where active = ?";
3
4    // prepara a consulta, por meio da conexão (conn)
5    PreparedStatement stmt = conn.prepareStatement(sql);
6
7    // configura os parâmetros da consulta
8    stmt.setInt(1, 0); // colocar 0 na primeira interrogação
9
10   int n = stmt.executeUpdate();
```

Note que o executeUpdate nos retorna um inteiro. Esse número indica quantas linhas foram alteradas, inseridas ou removidas da tabela em questão. É interessante verificar sempre esse valor. Assim, podemos ter uma ideia se o comando foi ou não executado com sucesso. Exemplo: se não tiverem “customers” inativos, o n será 0!

Fechando as Conexões

Como último passo da nossa “receita de bolo” para acesso a bancos de dados usando JDBC, é necessário finalizar, fechar a conexão com o banco de dados. Isso é importante porque, se não o fizermos, tanto o banco de dados como nosso programa passa a consumir recursos de máquina (memória, basicamente) de forma desnecessária.

Para fazer isso, a gente deve usar o método close() no objeto da classe Connection. No entanto, saiba que o ResultSet e o PreparedStatement também podem ser fechados. Ou seja, podemos deixar a conexão aberta, fechar apenas o conjunto de resultados, e reaproveitar a conexão aberta para fazer outra consulta. Aí, no final de tudo, deve-se fechar a conexão. Dessa forma, é importante seguir essa ordem no código:

```
1        rs.close();
2        stmt.close();
3        conn.close();
```

Tratando Erros

Como nem tudo são flores, muita coisa pode dar errado:

O “driver” pode não estar instalado e, ao se tentar carregá-lo por meio do comando Class.forName(driver);, uma exceção ClassNotFoundException pode ser gerada.

O banco pode estar “fora do ar” ou a rede pode estar com problemas. Nesse caso, ao se tentar uma conexão por meio do comando DriverManager.getConnection(url, user, pass), uma SQLException pode ser gerada.

Por conta desses mesmos erros citados no item anterior, a mesma exceção pode ser lançada pelos métodos prepareStatement(sql), executeQuery(), executeUpdate(), next(), getString(...), getInt(...) e afins, etc.

Dessa forma, o nosso código deve tratar essas exceções usando try/catch e, principalmente, fechar a conexão e os outros objetos pertinentes no bloco finally. Com isso, garantimos a liberação dos recursos de máquina usados. Note que devemos fazer isso no bloco finally porque ele é sempre executado, mesmo que uma exceção não tratada seja lançada dentro do bloco try. Vejamos um código completo:

```
1   try {
2       String driver = "com.mysql.jdbc.Driver";
3       String url = "jdbc:mysql://localhost:3306/sakila";
4       String user = "root";
5       String pass = "root";
6
7       String sql = "select * from customer order by first_name, last_name";
8
9       Class.forName(driver);
10      Connection conn = DriverManager.getConnection(url, user, pass);
11      PreparedStatement stmt = conn.prepareStatement(sql);
12      ResultSet rs = stmt.executeQuery();
13      while (rs.next()) {
14          String nome = rs.getString("first_name");
15          String sobrenome = rs.getString("last_name");
16          int ativo = rs.getInt("active");
17          int loja = rs.getInt("store_id");
18
19          System.out.println(loja + " - " + nome + " " + sobrenome + " - " + ativo);
```

```
20      }
21  } catch (ClassNotFoundException ex) {
22      System.out.println("Driver não encontrado!");
23  } catch (SQLException ex) {
24      System.out.println("Erro de banco: " + ex.getMessage());
25  } finally {
26      rs.close();
27      stmt.close();
28      conn.close();
29  }
```

Se você prestou atenção direitinho no código acima (ou se copiou-e-colou na sua IDE), deve ter notado que o bloco finally contém erros: as variáveis rs, stmt e conn não podem ser acessadas porque elas foram criadas dentro do bloco try, ou seja, em outro escopo. Para resolver isso, deve-se declará-las fora do bloco try/catch. Só que, com isso, precisamos verificar se essas variáveis não estão nulas e tratar a exceção que pode ser gerada pela chamada do método close(). Veja:

```
1   Connection conn = null;
2   PreparedStatement stmt = null;
3   ResultSet rs = null;
4
5   try {
6       String driver = "com.mysql.jdbc.Driver";
7       String url = "jdbc:mysql://localhost:3306/sakila";
8       String user = "root";
9       String pass = "root";
10
11      String sql = "select * from customer order by first_name, last_name";
12
13      Class.forName(driver);
14      conn = DriverManager.getConnection(url, user, pass);
15      stmt = conn.prepareStatement(sql);
```

```
16      rs = stmt.executeQuery();
17      while (rs.next()) {
18          String nome = rs.getString("first_name");
19          String sobrenome = rs.getString("last_name");
20          int ativo = rs.getInt("active");
21          int loja = rs.getInt("store_id");
22
23          System.out.println(loja + " - " + nome + " " + sobrenome + " - " + ativo);
24      }
25  } catch (ClassNotFoundException ex) {
26      System.out.println("Driver não encontrado!");
27  } catch (SQLException ex) {
28      System.out.println("Erro de banco: " + ex.getMessage());
29  } finally {
30      if (rs != null) { try { rs.close(); } catch (SQLException ex) {} }
31      if (stmt != null) { try { stmt.close(); } catch (SQLException ex) {} }
32      if (conn != null) { try { conn.close(); } catch (SQLException ex) {} }
33  }
```

Melhorando…

Existe uma outra forma de fazer esse tratamento de erros que evita que precisemos fazer toda essa verificação no bloco finally. O código está logo abaixo e a explicação no próprio link do artigo indicado…

```
1   try {
2       String driver = "com.mysql.jdbc.Driver";
3       String url = "jdbc:mysql://localhost:3306/sakila";
4       String user = "root";
5       String pass = "root";
6
7       String sql = "select * from customer order by first_name, last_name";
8
```

```
9       Class.forName(driver);
10      Connection conn = DriverManager.getConnection(url, user, pass);
11      try {
12          PreparedStatement stmt = conn.prepareStatement(sql);
13          try {
14              ResultSet rs = stmt.executeQuery();
15              try {
16                  while (rs.next()) {
17                      String nome = rs.getString("first_name");
18                      String sobrenome = rs.getString("last_name");
19                      int ativo = rs.getInt("active");
20                      int loja = rs.getInt("store_id");
21
22                      System.out.println(loja + " - " + nome + " " + sobrenome + " - " + ativo);
23                  }
24              } finally {
25                  rs.close();
26              }
27          } finally {
28              stmt.close();
29          }
30      } finally {
31          conn.close();
32      }
33  } catch (ClassNotFoundException ex) {
34      System.out.println("Driver não encontrado!");
35  } catch (SQLException ex) {
36      System.out.println("Erro de banco: " + ex.getMessage());
37  }
```

O código fica melhor, mas pode ser estranho para a maioria dos iniciantes…

Melhorando mais ainda!

A partir da versão 7, o Java possui uma nova forma de fazer tratamento de erros chamada “try-with-resources“. Basicamente, ele foi feito especificamente para esse tipo de código que estamos tratando: os que precisam fechar conexões, arquivos, etc. Ele funciona assim:

```
1    try (Recurso1 r1 = CriacaoDo.Recurso1();
2         Recurso2 r2 = CriacaoDo.Recurso2()) {
3        // uso dos recursos r1 e r2 que podem gerar exceção...
4    } catch (ExcecaoASerTratada ex) {
5        // tratamento do erro...
6    }
```

O que mudou? Bom, a criação dos objetos que precisam ser fechados vão dentro dos parênteses. O que tiver aí será fechado automaticamente!!! “Muuuito” melhor, não? Vamos ver, então, como fica o nosso código:

```
1    try {
2        String driver = "com.mysql.jdbc.Driver";
3        String url = "jdbc:mysql://localhost:3306/sakila";
4        String user = "root";
5        String pass = "root";
6
7        String sql = "select * from customer order by first_name, last_name";
8
9        Class.forName(driver);
10       try (Connection conn = DriverManager.getConnection(url, user, pass);
11            PreparedStatement stmt = conn.prepareStatement(sql);
12            ResultSet rs = stmt.executeQuery()) {
13           while (rs.next()) {
14               String nome = rs.getString("first_name");
15               String sobrenome = rs.getString("last_name");
16               int ativo = rs.getInt("active");
17               int loja = rs.getInt("store_id");
18
```

```
19              System.out.println(loja + " - " + nome + " " + sobrenome + " - " + ativo);
20          }
21      }
22  } catch (ClassNotFoundException ex) {
23      System.out.println("Driver não encontrado!");
24  } catch (SQLException ex) {
25      System.out.println("Erro de banco: " + ex.getMessage());
26  }
```

“Muuuuuuito” melhor!!! Mas, você deve estar se perguntando, e se o SQL tiver parâmetros? Bom, aí temos que separar o bloco try em dois: um antes de estabelecer os parâmetros por meio dos métodos setString(..., ...), setInt(..., ...), etc. e outro depois, só para o ResultSet. Assim:

```
1   try {
2       String driver = "com.mysql.jdbc.Driver";
3       String url = "jdbc:mysql://localhost:3306/sakila";
4       String user = "root";
5       String pass = "root";
6
7       String sql = "delete from customer where active = ?";
8
9       Class.forName(driver);
10      try (Connection conn = DriverManager.getConnection(url, user, pass);
11           PreparedStatement stmt = conn.prepareStatement(sql)) {
12          stmt.setInt(1, 0);
13          try (ResultSet rs = stmt.executeQuery()) {
14              while (rs.next()) {
15                  String nome = rs.getString("first_name");
16                  String sobrenome = rs.getString("last_name");
17                  int ativo = rs.getInt("active");
18                  int loja = rs.getInt("store_id");
19
```

```
20                System.out.println(loja + " - " + nome + " " + sobrenome + " - " + ativo);
21            }
22        }
23    }
24 } catch (ClassNotFoundException ex) {
25    System.out.println("Driver não encontrado!");
26 } catch (SQLException ex) {
27    System.out.println("Erro de banco: " + ex.getMessage());
28 }
```

Conclusão

Vimos que o JDBC é uma API que pode acessar bancos de dados de forma independente. Também vimos que ele faz essa “mágica” por meio do uso dos conceitos de Programação Orientada a Objetos. Aliás, você notou que o único local do nosso código onde foi citado o MySQL foi nas configurações, mais precisamente dentro de uma string que poderia ter sido lida de um arquivo de configuração? Ou seja, nosso código mesmo nem sabe que está acessando o MySQL. Poderia ser o Oracle, o PostgreSQL ou qualquer outro!!!

Além disso, vimos os passos básicos de uma “receita de bolo” para usar o JDBC:

Carrega o driver (Class.forName)

Faz a conexão (DriverManager.getConnection)

Prepara a consulta SQL (prepareStatement + setXXX)

Executa a consulta (executeQuery)

Itera pelos resultados (next + getXXX)

No caso de comandos SQL que atualizam a base por meio de insert, delete ou update, a receita tem menos passos:

Carrega o driver (Class.forName)

Faz a conexão (DriverManager.getConnection)

Prepara a consulta SQL (prepareStatement + setXXX)

Executa a consulta (executeUpdate)

Detalhe super importante que eu não comentei anteriormente, mas, se você estiver usando uma IDE, não precisou se preocupar: para usar as classes do JDBC é necessário fazer o import http://www.ramon.pro.br/jdbc-acessando-bancos-de-dados-em-java/do pacote java.sql. De qualquer maneira, os códigos estão disponíveis no GitHub.

http://www.ramon.pro.br/jdbc-acessando-bancos-de-dados-em-java/ acessado em 06/09/2018