Java e os Bancos de Dados

Namom Alves Alencar

Bancos de dados e JDBC

- O que é um Banco de dados?
- Porque usar?
- De que se alimentam?
- Como vivem?

Bancos de dados e JDBC

- Conectar-se a um banco de dados qualquer através da API JDBC;
- Criar uma fábrica de conexões usando o design pattern Factory;
- Pesquisar dados através de queries;
- Encapsular suas operações com bancos de dados através de DAO - Data Access Object.

Por que usar um banco de dados?

- Muitos sistemas precisam manter as informações com as quais eles trabalham, seja para permitir consultas futuras, geração de relatórios ou possíveis alterações nas informações. Para que esses dados sejam mantidos para sempre, esses sistemas geralmente guardam essas informações em um banco de dados, que as mantém de forma organizada e prontas para consultas.

- A maioria dos bancos de dados comerciais são os chamados relacionais, que é uma forma de trabalhar e pensar diferente ao paradigma orientado a objetos.

Por que usar um banco de dados?

- O MySQL é o banco de dados que usaremos durante o curso. É um dos mais importantes bancos de dados relacionais, e é gratuito, além de ter uma instalação fácil para todos os sistemas operacionais. Depois de instalado, para acessá-lo via terminal, fazemos da seguinte forma:

- mysql -u root

Por que usar um banco de dados?

- O processo de armazenamento de dados é também chamado de persistência. A biblioteca de persistência em banco de dados relacionais do Java é chamada JDBC, e também existem diversas ferramentas do tipo ORM (Object Relational Mapping) que facilitam bastante o uso do JDBC. Neste momento, focaremos nos conceitos e no uso do JDBC.
- No final do curso posso mostrar um poquinho do JPA Hibernate.

Persistindo através de Sockets?

- Para conectar-se a um banco de dados poderíamos abrir sockets diretamente com o servidor que o hospeda, por exemplo um Oracle ou MySQL e nos comunicarmos com ele através de seu protocolo proprietário.

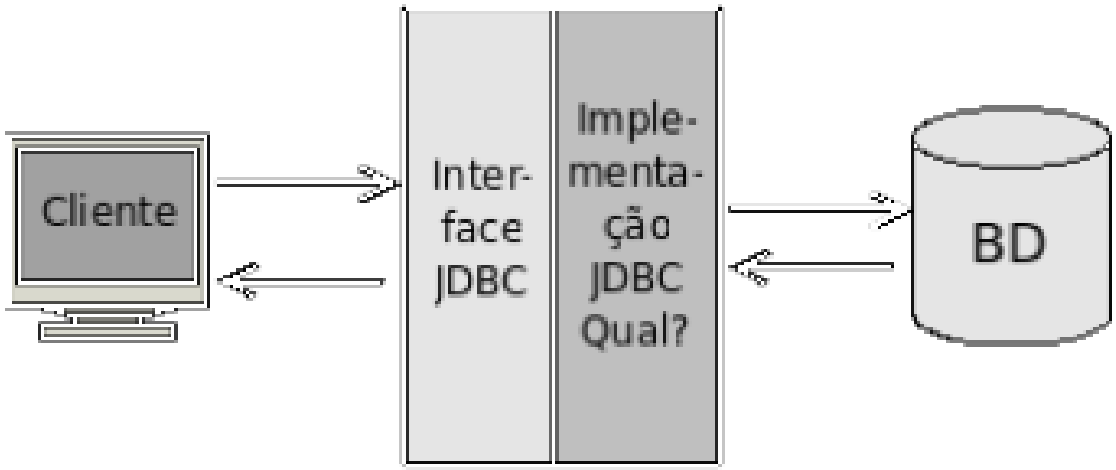
- Mas você conhece o protocolo proprietário de algum banco de dados? Conhecer um protocolo complexo em profundidade é difícil, trabalhar com ele é muito trabalhoso.

Persistindo através de Sockets?

- Uma segunda ideia seria utilizar uma API específica para cada banco de dados. Antigamente, no PHP, por exemplo, a única maneira de acessar o Oracle era através de funções como oracle_connect, oracle_result, e assim por diante. O MySQL tinha suas funções análogas, como mysql_connect. Essa abordagem facilita muito nosso trabalho por não precisarmos entender o protocolo de cada banco, mas faz com que tenhamos de conhecer uma API um pouco diferente para cada tipo de banco. Além disso, caso precisemos trocar de banco de dados um dia, precisaremos trocar todo o nosso código para refletir a função correta de acordo com o novo banco de dados que estamos utilizando.

A conexão em Java

- Conectar-se a um banco de dados com Java é feito de maneira elegante. Para evitar que cada banco tenha a sua própria API e conjunto de classes e métodos, temos um único conjunto de interfaces muito bem definidas que devem ser implementadas. Esse conjunto de interfaces fica dentro do pacote java.sql e nos referiremos a ela como JDBC.



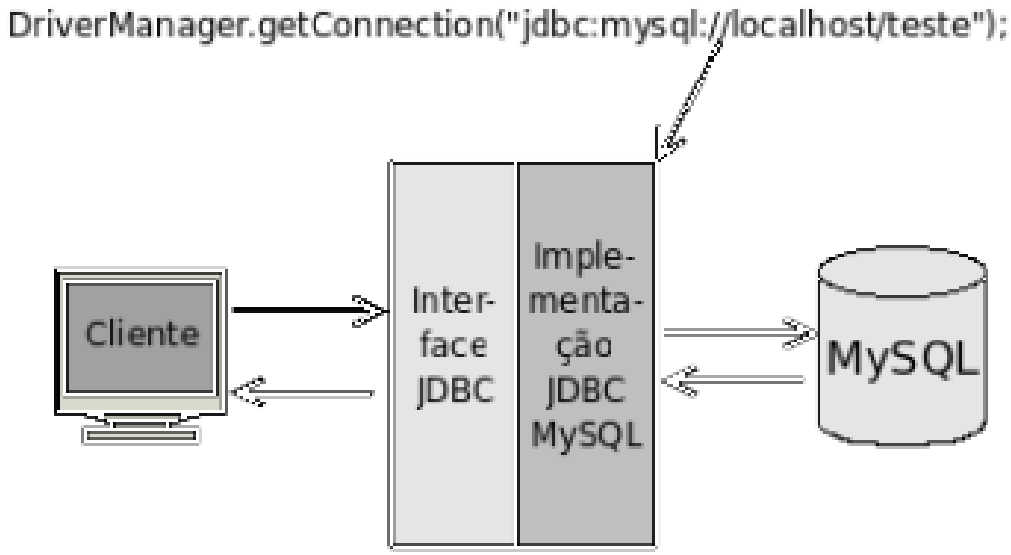
A conexão em Java

- Entre as diversas interfaces deste pacote, existe a interface Connection que define métodos para executar uma query (como um insert e select), comitar transação e fechar a conexão, entre outros. Caso queiramos trabalhar com o MySQL, precisamos de classes concretas que implementem essas interfaces do pacote java.sql.

A conexão em Java

- Esse conjunto de classes concretas é quem fará a ponte entre o código cliente que usa a API JDBC e o banco de dados. São essas classes que sabem se comunicar através do protocolo proprietário do banco de dados. Esse conjunto de classes recebe o nome de driver. Todos os principais bancos de dados do mercado possuem drivers JDBC para que você possa utilizá-los com Java. O nome driver é análogo ao que usamos para impressoras: como é impossível que um sistema operacional saiba conversar com todo tipo de impressora existente, precisamos de um driver que faça o papel de "tradutor" dessa conversa.

A conexão em Java



- Para abrir uma conexão com um banco de dados, precisamos utilizar sempre um driver. A classe DriverManager é a responsável por se comunicar com todos os drivers que você deixou disponível. Para isso, invocamos o método estático getConnection com uma String que indica a qual banco desejamos nos conectar.

A conexão em Java

- Seguindo o exemplo da linha acima e tudo que foi dito até agora, seria possível rodar o exemplo abaixo e receber uma conexão para um banco MySQL, caso ele esteja rodando na mesma máquina:

```
public class JDBCExemplo {
    public static void main(String[] args) throws SQLException {
        Connection conexao = DriverManager.getConnection(
          "jdbc:mysql://localhost/banco_curso_java");
        System.out.println("Conectado!");
        conexao.close();
    }
}
```

A conexão em Java

- Repare que estamos deixando passar a SQLException, que é uma exception checked, lançada por muitos dos métodos da API de JDBC. Numa aplicação real devemos utilizar try/catch nos lugares que julgamos haver possibilidade de recuperar de uma falha com o banco de dados. Também precisamos tomar sempre cuidado para fechar todas as conexões que foram abertas.

A conexão em Java

- Ao testar o código acima, recebemos uma exception. A conexão não pôde ser aberta. Recebemos a mensagem:

java.sql.SQLException: No suitable driver found for

jdbc:mysql:// localhost/banco_curso_java

Por que?

A conexão em Java

- O sistema ainda não achou uma implementação de driver JDBC que pode ser usada para abrir a conexão indicada pela URL
  - jdbc:mysql://localhost/banco_curso_java
- O que precisamos fazer é adicionar o driver do MySQL ao classpath, o arquivo .jar contendo a implementação JDBC do MySQL (mysql connector) precisa ser colocado em um lugar visível pelo seu projeto ou adicionado à variável de ambiente CLASSPATH. Como usaremos o Eclipse, fazemos isso através de um clique da direita em nosso projeto, Properties/Java Build Path e em Libraries adicionamos o jar do driver JDBC do MySQL.

E o Class.forName?

- Até a versão 3 do JDBC, antes de chamar o DriverManager.getConnection() era necessário registrar o driver JDBC que iria ser utilizado através do método Class.forName("com.mysql.jdbc.Driver"), no caso do MySQL, que carregava essa classe, e essa se comunicava com o DriverManager.

E o Class.forName?

- A partir do JDBC 4, que está presente no Java 6, esse passo não é mais necessário. Mas lembre-se: caso você utilize JDBC em um projeto com Java 5 ou anterior, será preciso fazer o registro do Driver JDBC, carregando a sua classe, que vai se registrar no DriverManager.

- Isso também pode ser necessário em alguns servidores de aplicação e web, como no Tomcat 7(mais utilizado) ou posterior, por proteção para possíveis vazamentos de memória:

E o Class.forName?

- A partir do JDBC 4, que está presente no Java 6, esse passo não é mais necessário. Mas lembre-se: caso você utilize JDBC em um projeto com Java 5 ou anterior, será preciso fazer o registro do Driver JDBC, carregando a sua classe, que vai se registrar no DriverManager.

- Isso também pode ser necessário em alguns servidores de aplicação e web, como no Tomcat 7(mais utilizado) ou posterior, por proteção para possíveis vazamentos de memória:

Alterando o banco de dados

- Teoricamente, basta alterar as duas Strings que escrevemos para mudar de um banco para outro. Porém, não é tudo tão simples assim! Alguns códigos SQL que veremos a seguir pode funcionar em um banco e não em outros. Depende de quão aderente ao padrão ANSI SQL é seu banco de dados.
- Isso só causa dor de cabeça e existem projetos que resolvem isso, como é o caso do Hibernate (www.hibernate.org) e da especificação JPA (Java Persistence API).

Fábrica de Conexões

- Em determinado momento de nossa aplicação, gostaríamos de ter o controle sobre a construção dos objetos da nossa classe. Muito pode ser feito através do construtor, como saber quantos objetos foram instanciados ou fazer o log sobre essas instanciações.

- Às vezes, também queremos controlar um processo muito repetitivo e trabalhoso, como abrir uma conexão com o banco de dados. Tomemos como exemplo a classe a seguir que seria responsável por abrir uma conexão com o banco:

Fábrica de Conexões

```
public class ConnectionFactory {
    public Connection getConnection() {
        try {
            return DriverManager.getConnection(
          "jdbc:mysql://localhost/banco_curso_java", "root", "");
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Fábrica de Conexões

- Poderíamos colocar um aviso na nossa aplicação, notificando todos os programadores a adquirir uma conexão:

```
Connection con = new ConnectionFactory().getConnection();
```

- Note que o método getConnection() é uma fábrica de conexões, isto é, ele cria novas conexões para nós. Basta invocar o método e recebemos uma conexão pronta para uso, não importando de onde elas vieram e eventuais detalhes de criação. Portanto, vamos chamar a classe de ConnectionFactory e o método de getConnection.

Fábrica de Conexões

- Encapsulando dessa forma, podemos mais tarde mudar a obtenção de conexões, para, por exemplo, usar um mecanismo de *pooling*, que é fortemente recomendável em uma aplicação real.

- Quando falarmos de JSF podemos falar um pouco mais de pooling de conexões.

Design Patterns

- Orientação a objetos resolve as grandes dores de cabeça que tínhamos na programação procedural, restringindo e centralizando responsabilidades.

- Mas alguns problemas não podemos simplesmente resolver com orientação a objetos, pois não existe palavra chave para uma funcionalidade tão específica.

Design Patterns

- Alguns desses pequenos problemas aparecem com tanta frequência que as pessoas desenvolvem uma solução "padrão" para ele. Com isso, ao nos defrontarmos com um desses problemas clássicos, podemos rapidamente implementar essa solução genérica com uma ou outra modificação, de acordo com nossa necessidade. Essa solução padrão tem o nome de Design Pattern (Padrão de Projeto).

Design Patterns

- A melhor maneira para aprender o que é um Design Pattern é vendo como surgiu sua necessidade.

- A nossa ConnectionFactory implementa o design pattern Factory que prega o encapsulamento da construção (fabricação) de objetos complicados.

Design Patterns

- A bíblia dos Design Patterns

- O livro mais conhecido de Design Patterns foi escrito em 1995 e tem trechos de código em C++ e Smalltalk. Mas o que realmente importa são os conceitos e os diagramas que fazem desse livro independente de qualquer linguagem. Além de tudo, o livro é de leitura agradável.

Exercícios: ConnectionFactory

- 1 – Criar uma fábrica de conexões;
  - Novo Projeto
  - Novo pacote br.com.cursojava.jdbc.conexao
  - Baixar a ultima versão do Drive do MYSQL
  - Coloque o drive dentro do seu workspace
  - Criar uma classe chamada ConnectionFactor
  - Cuidado ao importar(sempre importe java.sql)

Exercícios: ConnectionFactory

- 2 - Crie uma classe chamada TestaConexao no pacote br.com.cursojava.jdbc.teste. Todas as nossas classes de teste deverão ficar nesse pacote.
  - Dentro do main, fabrique uma conexão usando a ConnectionFactory que criamos. Vamos apenas testar a abertura da conexão e depois fechá-la com o método close:

```
Connection connection = new ConnectionFactory().getConnection();
System.out.println("Conexão aberta!");
connection.close();
```

  - Trate os erros com throws. ("add throws declaration").
  - **Funcionou? Qual foi o problema?**

Exercícios: ConnectionFactory

- 3 - Parece que a aplicação não funciona pois o driver não foi encontrado? Esquecemos de colocar o JAR no classpath! (Build Path no Eclipse)
  - Clique no seu projeto com o botão da direita e escolha Refresh (ou pressione F5).
  - Selecione o seu driver do MySQL, clique da direita e escolha Build Path, Add to Build Path.
  - Rode novamente sua aplicação TestaConexao agora que colocamos o driver no classpath.

A tabela Contato

- Para criar uma tabela nova, primeiro devemos acessar o terminal e fazermos o comando para logarmos no mysql.
- mysql -u root
- Vamos criar um banco de dados
  - Create database banco_curso_java;
  - Use banco_curso_java;
- Não se esqueça dos “;”

A tabela Contato

- Criaremos a seguinte tabela

```
Create table contatos (
  id BIGINT NOT NULL AUTO_INCREMENT,
  nome VARCHAR(255),
  email VARCHAR(255),
  endereco VARCHAR(255),
  dataNascimento DATE,
  primary key (id)
);
```

- No banco de dados relacional, é comum representar um contato (entidade) em uma tabela de contatos.

Javabeans

- O que são Javabeans? A pergunta que não quer se calar pode ser respondida muito facilmente uma vez que a uma das maiores confusões feitas aí fora é entre Javabeans e Enterprise Java Beans (EJB).

- Javabeans são classes que possuem o construtor sem argumentos e com métodos de acesso do tipo get e set! Mais nada! Simples, não? Já os EJBs costumam ser javabeans com características mais avançadas. Por exemplo, facilidade de deploy. Entity...(Grandes Empresas)

Javabeans

- Podemos usar beans por diversos motivos, normalmente as classes que representam nosso modelo de dados são usados dessa forma.

- Utilizaremos:
- uma classe com métodos do tipo get e set para cada um de seus parâmetros, que representa algum objeto;

- uma classe com construtor sem argumentos que representa uma coleção de objetos.

Javabeans

- Podemos usar beans por diversos motivos, normalmente as classes que representam nosso modelo de dados são usados dessa forma.

- Utilizaremos:
- uma classe com métodos do tipo get e set para cada um de seus parâmetros, que representa algum objeto;

- uma classe com construtor sem argumentos que representa uma coleção de objetos.

Javabeans

- Uma classe JavaBean que seria equivalente ao nosso modelo de entidade do banco de dados:

```
public class Contato {
  private Long id;
  private String nome;
  private String email;
  private String endereco;
  private Calendar dataNascimento;
  // métodos get e set para id, nome, email, endereço e dataNascimento
}
```

Inserindo dados no banco

- Para inserir dados em uma tabela de um banco de dados entidade-relacional basta usar a cláusula INSERT. Precisamos especificar quais os campos que desejamos atualizar e os valores.
- Primeiro o código SQL:

```
String sql = "insert into contatos " +
        "(nome,email,endereco, dataNascimento)" +
        " values ('" + nome + "', '" + email + "', '" +
        endereco + "', '"+ dataNascimento +"')";
```

Inserindo dados no banco

- O exemplo acima possui três pontos negativos que são importantíssimos. O primeiro é que o programador que não escreveu o código original não consegue bater o olho e entender o que está escrito. O que o código acima faz? Lendo rapidamente fica difícil. Mais difícil ainda é saber se faltou uma vírgula, um fecha parênteses talvez? E ainda assim, esse é um caso simples. Existem tabelas com 10, 20, 30 e até mais campos, tornando inviável entender o que está escrito no SQL misturado com as concatenações.

Inserindo dados no banco

- Outro problema é o clássico "preconceito contra Joana D'arc", formalmente chamado de SQL Injection. O que acontece quando o contato a ser adicionado possui no nome uma aspas simples? O código SQL se quebra todo e para de funcionar ou, pior ainda, o usuário final é capaz de alterar seu código sql para executar aquilo que ele deseja (SQL injection)... tudo isso porque escolhemos aquela linha de código e não fizemos o escape de caracteres especiais.

Inserindo dados no banco

- Mais um problema que enxergamos aí é na data. Ela precisa ser passada no formato que o banco de dados entenda e como uma String, portanto, se você possui um objeto java.util.Calendar que é o nosso caso, você precisará fazer a conversão desse objeto para a String.

Inserindo dados no banco

- Por esses três motivos não usaremos código SQL como mostrado anteriormente. Vamos imaginar algo mais genérico e um pouco mais interessante:

```
String sql = "insert into contatos " +
        "(nome,email,endereco,dataNascimento) " +
        "values (?,?,?,?)";
```

Inserindo dados no banco

- Perceba que não colocamos os pontos de interrogação de brincadeira, mas sim porque realmente não sabemos o que desejamos inserir. Estamos interessados em executar aquele código mas não sabemos ainda quais são os parâmetros que utilizaremos nesse código SQL que será executado, chamado de statement.
- As cláusulas são executadas em um banco de dados através da interface PreparedStatement. Para receber um PreparedStatement relativo à conexão, basta chamar o método prepareStatement, passando como argumento o comando SQL com os valores vindos de variáveis preenchidos com uma interrogação.

Inserindo dados no banco

```
String sql = "insert into contatos " +
             "(nome,email,endereco,dataNascimento) " +
             "values (?,?,?,?)";
PreparedStatement stmt = connection.prepareStatement(sql);
```

- Logo em seguida, chamamos o método setString do PreparedStatement para preencher os valores que são do tipo String, passando a posição (começando em 1) da interrogação no SQL e o valor que deve ser colocado:

```
stmt.setString(1, “CursoJava");
stmt.setString(2, “namom@namomalencar.com");
stmt.setString(3, “Rua Feliz, 500 Fortaleza CE");
```

E a data?

Inserindo dados no banco

- Precisamos definir também a data de nascimento do nosso contato, para isso, precisaremos de um objeto do tipo java.sql.Date para passarmos para o nosso PreparedStatement. Nesse exemplo, vamos passar a data atual. Para isso, vamos passar um long que representa os milissegundos da data atual para dentro de um java.sql.Date que é o tipo suportado pela API JDBC. Vamos utilizar a classe Calendar para conseguirmos esses milissegundos:

```
java.sql.Date dataParaGravar = new java.sql.Date(
        Calendar.getInstance().getTimeInMillis());
stmt.setDate(4, dataParaGravar);
```

Inserindo dados no banco

- Por fim, uma chamada a execute() executa o comando SQL:

```
stmt.execute();
```

- Imagine todo esse processo sendo escrito toda vez que desejar inserir algo no banco? Ainda não consegue visualizar o quão destrutivo isso pode ser?

Fechando a conexão propriamente

```
public class JDBCInsere {
   public static void main(String[] args) throws SQLException {
   Connection con = new ConnectionFactory().getConnection();
   String sql = "insert into contatos" +
            " (nome,email,endereco,dataNascimento)" +
            " values (?,?,?,?)";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, “CursoJava");
      stmt.setString(2, “namom@namom.com");
      stmt.setString(3, "R. Feliz, 500 Fortaleza CE");
      stmt.setDate(4, new java.sql.Date(
            Calendar.getInstance().getTimeInMillis()));
    stmt.execute();
      stmt.close();
      System.out.println("Gravado!");
      con.close();
   } }
```

- Para inserir uma linha olha como fica gigante o código.

Inserindo dados no banco

- Não é comum utilizar JDBC diretamente hoje em dia. O mais praticado é o uso de alguma API de ORM como a JPA ou o Hibernate. Tanto na JDBC quanto em bibliotecas ORM deve-se prestar atenção no momento de fechar a conexão.

- O exemplo dado acima não a fecha caso algum erro ocorra no momento de inserir um dado no banco de dados. O comum é fechar a conexão em um bloco finally:

Inserindo dados no banco

```
public class JDBCInsere {
    public static void main(String[] args) throws SQLException {
      Connection con = null;
   try {
          con = new ConnectionFactory().getConnection();
    // faz um monte de operações.
    // que podem lançar exceptions runtime e SQLException
   catch(SQLException e) {
    System.out.println(e);
   } finally {
          con.close();
        }    }  }
```

Inserindo dados no banco

- Dessa forma, mesmo que o código dentro do try lance exception, o con.close() será executado. Garantimos que não deixaremos uma conexão pendurada sem uso. Esse código pode ficar muito maior se quisermos ir além. Pois o que acontece no caso de con.close lançar uma exception? Quem a tratará?

- Trabalhar com recursos caros, como conexões, sessões, threads e arquivos, sempre deve ser muito bem pensado. Deve-se tomar cuidado para não deixar nenhum desses recursos abertos, pois poderá vazar algo precioso da nossa aplicação. Como veremos durante o curso, é importante centralizar esse tipo de código em algum lugar, para não repetir uma tarefa complicada como essa.

Inserindo dados no banco

- Além disso, há a estrutura do Java 7 conhecida como try-with-resources. Ela permite declarar e inicializar, dentro do try, objetos que implementam AutoCloseable. Dessa forma, ao término do try, o próprio compilador inserirá instruções para invocar o close desses recursos, além de se precaver em relação a exceções que podem surgir por causa dessa invocação. Nosso código ficaria mais reduzido e organizado, além do escopo de con só valer dentro do try:

Inserindo dados no banco

```
try(Connection con = new ConnectionFactory().getConnection()) {
  // faz um monte de operações.
  // que podem lançar exceptions runtime e SQLException
} catch(SQLException e) {
  System.out.println(e);
}
```

JodaTime

- A API de datas do Java, mesmo considerando algumas melhorias da Calendar em relação a Date, ainda é muito pobre. Na versão 8 do Java temos uma API nova, a java.time, que facilita bastante o trabalho. Ela é baseada na excelente biblioteca de datas chamada JodaTime.

DAO - Data Access Object

- Já foi possível sentir que colocar código SQL dentro de suas classes de lógica é algo nem um pouco elegante e muito menos viável quando você precisa manter o seu código.

- É tenso pegar um código cheio de SQL, esse tipo de código é quase ilegivel.

DAO - Data Access Object

- A ideia é remover o código de acesso ao banco de dados de suas classes de lógica e colocá-lo em uma classe responsável pelo acesso aos dados.

- Assim o código de acesso ao banco de dados fica em um lugar só, tornando mais fácil a manutenção.

- Que tal se pudéssemos chamar um método adiciona que adiciona um Contato ao banco?

DAO - Data Access Object

- Em outras palavras quero que o código a seguir funcione:

```
// adiciona os dados no banco
Misterio bd = new Misterio();
bd.adiciona("meu nome", "meu email", "meu endereço",
meuCalendar);
```

DAO - Data Access Object

- Tem algo estranho nesse código. Repare que todos os parâmetros que estamos passando são as informações do contato. Se contato tivesse 20 atributos, passaríamos 20 parâmetros? Java é orientado a Strings? Vamos tentar novamente: em outras palavras quero que o código a seguir funcione:

```
// adiciona um contato no banco
Misterio bd = new Misterio();
// método muito mais elegante
bd.adiciona(contato);
```

DAO - Data Access Object

- Tentaremos chegar ao código anterior: seria muito melhor e mais elegante poder chamar um único método responsável pela inclusão, certo?

```
public class TestaInsere {
  public static void main(String[] args) {
     // pronto para gravar
     Contato contato = new Contato();
     contato.setNome(“Namom Curso Java"); ///...sets
     // grave nessa conexão!!!
     Misterio bd = new Misterio();
     // método elegante
     bd.adiciona(contato);
     System.out.println("Gravado!");  } }
```

DAO - Data Access Object

- O código anterior já mostra o poder que alcançaremos: através de uma única classe seremos capazes de acessar o banco de dados e, mais ainda, somente através dessa classe será possível acessar os dados.

- Esta ideia, inocente à primeira vista, é capaz de isolar todo o acesso a banco em classes bem simples, cuja instância é um objeto responsável por acessar os dados. Da responsabilidade deste objeto surgiu o nome de Data Access Object ou simplesmente DAO, um dos mais famosos padrões de projeto (design pattern).

DAO - Data Access Object

- O que falta para o código acima funcionar é uma classe chamada ContatoDao com um método chamado adiciona. Vamos criar uma que se conecta ao banco ao construirmos uma instância dela:

```
public class ContatoDao {
  // a conexão com o banco de dados
  private Connection connection;
  public ContatoDao() {
    this.connection = new ConnectionFactory().getConnection();
  }  }
```

DAO - Data Access Object

- Agora que todo ContatoDao possui uma conexão com o banco, podemos focar no método adiciona, que recebe um Contato como argumento e é responsável por adicioná-lo através de código SQL:

DAO - Data Access Object

```
public void adiciona(Contato contato) {
   String sql = "insert into contatos " +
         "(nome,email,endereco,dataNascimento) values (?,?,?,?)";
   try {       // prepared statement para inserção
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1,contato.getNome());
      stmt.setString(2,contato.getEmail());
      stmt.setString(3,contato.getEndereco());
      stmt.setDate(4, new Date(
            contato.getDataNascimento().getTimeInMillis()));
      stmt.execute();
      stmt.close();
   } catch (SQLException e) {
      throw new RuntimeException(e);
   } }
```

Exercícios: Javabeans e ContatoDao

- 1 - Crie a classe de Contato no pacote br.com.cursojava.jdbc.model. Ela será nosso JavaBean para representar a entidade do banco. Deverá ter id, nome, email, endereço e uma data de nascimento.

- 2 - Vamos desenvolver nossa classe de DAO. Crie a classe ContatoDao no pacote br.com.cursojava.dao. Seu papel será gerenciar a conexão e inserir Contatos no banco de dados.

- 3 – Faça o método adiciona no nosso ContatoDAO.

Exercícios: Javabeans e ContatoDao

- 4 - Para testar nosso DAO, desenvolva uma classe de testes com o método main. Por exemplo, uma chamada TestaInsere no pacote br.com.cursojava.teste.
- 5 – Abra o terminal do banco de dados e verifique se o contato foi adicionado.
  - Select * from contatos.

Fazendo pesquisas no banco de dados

- Show, legal. Mas agora precisamos buscar algo no banco de dados.

- Para pesquisar também utilizamos a interface PreparedStatement para montar nosso comando SQL. Mas como uma pesquisa possui um retorno (diferente de uma simples inserção), usaremos o método executeQuery que retorna todos os registros de uma determinada query.

- O objeto retornado é do tipo ResultSet do JDBC, o que nos permite navegar por seus registros através do método next. Esse método retornará false quando chegar ao fim da pesquisa, portanto ele é normalmente utilizado para fazer um laço nos registros:

Fazendo pesquisas no banco de dados

```
// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");
// executa um select
ResultSet rs = stmt.executeQuery();
// itera no ResultSet
while (rs.next()) {
}
rs.close();
stmt.close();
con.close();
```

Fazendo pesquisas no banco de dados

- Para retornar o valor de uma coluna no banco de dados, basta chamar um dos métodos get do ResultSet, dentre os quais, o mais comum: getString.

Fazendo pesquisas no banco de dados

```
// pega a conexão e o Statement
Connection con = new ConnectionFactory().getConnection();
PreparedStatement stmt = con.prepareStatement("select * from contatos");
// executa um select
ResultSet rs = stmt.executeQuery();
// itera no ResultSet
while (rs.next()) {
  String nome = rs.getString("nome");
  String email = rs.getString("email")
  System.out.println(nome + " :: " + email);
}
stmt.close();
con.close();
```

Fazendo pesquisas no banco de dados

- Mas, novamente, podemos aplicar as ideias de DAO e criar um método getLista() no nosso ContatoDao. Mas o que esse método retornaria? Um ResultSet? E teríamos o código de manipulação de ResultSet espalhado por todo o código? Vamos fazer nosso getLista() devolver algo mais interessante, uma lista de Contato:

Fazendo pesquisas no banco de dados

- Mas, novamente, podemos aplicar as ideias de DAO e criar um método getLista() no nosso ContatoDao. Mas o que esse método retornaria? Um ResultSet? E teríamos o código de manipulação de ResultSet espalhado por todo o código? Vamos fazer nosso getLista() devolver algo mais interessante, uma lista de Contato:

Exercícios: Listagem

- 1 - Crie o método getLista na classe ContatoDao. Importe List de java.util

- 2 - Vamos usar o método getLista para listar todos os contatos do nosso banco de dados.
  - Crie uma classe chamada TestaLista

- 3 - A impressão da data de nascimento ficou um pouco estranha. Para formatá-la, pesquise sobre a classe SimpleDateFormat.

- 4 - Crie uma classe chamada DAOException que estenda de RuntimeException e utilize-a no seu ContatoDao.

Cláusura WHERE

- Podemos filtrar colunas para nos mostrar apenas os dados que nos interessa através da cláusula *where* em conjunto com os operadores comparativos.
  - Select * from contatos where nome=“Namom”.
  - Select * from contatos where nome like “%am%”.
  - Select * from contatos where id = 123.

Cláusura WHERE

- 5 - Use cláusulas where para refinar sua pesquisa no banco de dados. Por exemplo: where nome like 'C%‘

- 6 - Crie o método pesquisar que recebe um id (int) e retorna um objeto do tipo Contato.

Outros métodos para o seu DAO

- Agora que você já sabe usar o PreparedStatement para executar qualquer tipo de código SQL e ResultSet para receber os dados retornados da sua pesquisa fica simples, porém maçante, escrever o código de diferentes métodos de uma classe típica de DAO.

- Como ficaria o SQL?

Outros métodos para o seu DAO

- Alterando um dado
  - Update contatos set nome=“Namom” where id=500.
- Como ficaria no prepareStatement
  - update contatos set nome=?, email=?, endereco=?, "dataNascimento=? where id=?
    - …..

Outros métodos para o seu DAO

- E para remover um dado?
  - Delete from contatos where nome= “Namom”
- Como ficaria no prepareStatement
  - Delete from contato where nome=?
  - ....

Exercicios

- 7 - Crie o método altera para redefir seu contato.
- 8 - Crie o remove para deletar seu contato.
- 9 – Teste na sua classe de testes.

Exercicios para casa

- Crie uma tabela chamada telefone com
  - Id
  - Telefone
  - Id_contato
- Crie uma classe chamada telefone que possui
  - String telefone
  - Contato contato
- Insira, altere, busque e remova do banco de dados.

Bons Estudos

Namom Alves Alencar