Transcrição

[00:00] Olá aluno, tudo bom? Hoje em dia, ao pensar em sistemas, a maioria deles são provedores de serviços. Pensando aqui na Alura, que é uma provedora de cursos online, no momento em que você vai fazer a matrícula, é necessário selecionar os campos de texto na página da Alura e inserir algumas informações para que você de fato tenha acesso aos seus cursos online.

[00:30] Então, preenchendo aqui o formulário que eles solicitam, eu vou agora selecionar, vou inserir qual é a minha informação de pagamento, então meu cartão de crédito, o nome que está no meu cartão de crédito, o código de segurança. Uma vez que eu termine de preencher as minhas informações corretamente, eu concluo o pagamento e vou ter acesso aos cursos.

[01:05] Uma coisa a se atentar aqui é sobre as informações que nós estamos fornecendo para a Alura e uma atenção, porque são informações bem críticas, ou seja, eu estou informando o número do meu cartão de crédito, estou informando o meu CPF. Essas informações, na mão de uma pessoa que não seja confiável, pode te trazer um prejuízo financeiro muito grande.

[01:32] Então onde salvar essas informações? Onde eu persisto essas informações de forma segura? Hoje em dia, o mais comum que tem no mercado são as bases de dados relacionais. Falando em banco de dados relacionais, nós temos muitos, nós temos o PostgreSQL, nós temos o SQL Server, que é da Microsoft, e nós temos o MySQL, que hoje eu me arrisco a dizer que é o mais famoso entre os bancos de dados.

[02:00] E vai ser ele que nós vamos utilizar para as próximas aulas, vamos utilizar ele como o nosso banco de dados do nosso ambiente. Em "mysql.com/downloads/" - lembrando que o site é "mysql.com", nós vamos rolar a página para baixo e vamos

ter um link "MySQL Community (GLP) Downloads". É nele que nós vamos clicar. Eu vou baixar o "MySQL Installer for Windows".

[02:40] Para nós não termos problemas de compatibilidade, nós vamos utilizar a versão 8.0.18. Você, que está chegando para fazer o curso, se tiver novas versões, mesmo assim, eu te peço encarecidamente para você manter na versão 8.0.18. Isso evita que, mais na frente, que nós vamos utilizar Querys mais avançadas, enfim, que não tenha nenhum problema, que todo mundo consiga fazer todos os comandos que nós vamos trabalhar nas aulas.

[03:23] Então tem duas opções de download e nós vamos fazer o download da segunda opção, que é a "mysql-installer-community-8.0.18.msi". Vocês podem instalar no seu diretório de preferência. Eu vou botar a minha na pasta "Downloads". Na verdade, eu já fiz o download do instalador. E agora eu vou na pasta "Downloads" e nós vamos de fato instalar o MySQL na nossa máquina.

[03:57] O MySQL, ele está aqui. Clico duas vezes no arquivo e vamos instalar. Agora que todas as configurações foram aplicadas, nós vamos finalizar a instalação do nosso MySQL. Muito bom, "configuração completa". Dou um "Next" e "Finish". Agora, a intenção é testar o nosso banco de dados, é nós criarmos Database, criarmos tabelas. E nós vamos fazer isso agora.

[05:23] No momento da instalação, o MySQL nos provê um MySQL 8.0 Command Line Client no menu "Iniciar" do Windows. É ele que nós vamos utilizar. Para ficar melhor a visualização para você, aluno, eu vou, clico com o botão direito sobre a barra de título do prompt de comando, vou em "Propriedades", vou mudar a fonte para "Lucida Console", vou botar o tamanho da fonte em "16".

[05:45] Em "Layout", eu vou aumentar um pouco essa janela, vou botar 150x80, vamos ver se vai ficar bom. Ficou bom. Agora a nossa senha padrão root. Estamos já interagindo com o MySQL. Ele bota as versões como eu informei para vocês, 8.0.18 é a que nós vamos utilizar no curso. Mesmo que tenha mais nova, vamos manter essa versão.

[06:23] A primeira coisa que nós vamos fazer é criar uma Database para nós. Então vou botar `CREATE DATABASE` e o nosso curso vai ter uma loja virtual, então `CREATE DATABASE loja_virtual;` . Criei a nossa Database, ele retornou um "Query ok",

então Database criada com sucesso.

[06:51] Eu vou colocar um `USE loja_virtual;` e agora nós mudamos para a nossa Database que acabamos de criar. A nossa Database está criada. Agora eu quero que tenha uma tabela chamada produto, que vai ter um produto nessa nossa loja virtual, e esse produto, eu quero que ele tenha um Id, um nome e uma descrição.

[07:16] Então nós vamos fazer um `CREATE TABLE PRODUTO` , ele vai ter um Id, que vai ser o inteiro `AUTO_INCREMENT` , ficando `CREATE TABLE PRODUTO (id INT AUTO_INCREMENT,` . Isso quer dizer que a cada nova inserção de um produto, o Id, ele vai ser inserido dinamicamente, então o primeiro produto o Id vai ser 1, o segundo Id vai ser 2, e assim sucessivamente.

[07:44] O `nome` , ele é uma string, ele é alguns caracteres, então no nosso banco de dados, isso é um `VARCHAR` , e eu quero um `VARCHAR` de 50 caracteres, e o `nome` também não vai poder ser nulo, ou seja, sempre vou ter que informar o meu nome. Então a linha fica `AUTO_INCREMENT` , ficando `CREATE TABLE PRODUTO (id INT AUTO_INCREMENT, nome VARCHAR(50) NOT NULL,` .

[08:04] `descricao` , descrição também vai ser um `VARCHAR` 255 - espera que ficou faltando o `VARCHAR` . E a nossa chave primária, a nossa `PRIMARY KEY` vai ser o Id. Então `AUTO_INCREMENT` , ficando `CREATE TABLE PRODUTO (id INT AUTO_INCREMENT, nome VARCHAR(50) NOT NULL, descricao VARCHAR(255), PRIMARY KEY (id))` .

[08:34] E vou botar um `Engine = InnoDB;` , que significa que eu quero que seja aceita a transação, ficando `AUTO_INCREMENT` , ficando `CREATE TABLE PRODUTO (id INT AUTO_INCREMENT, nome VARCHAR(50) NOT NULL, descricao VARCHAR(255), PRIMARY KEY (id)) Engine = InnoDB;` . Ele deu um erro, deixa eu ver o que foi. Isso, não tinha vírgula após `PRIMARY KEY (id))` .

[08:53] Criamos a nossa tabela. Se eu fizer agora um `SELECT * FROM PRODUTO;` , nós vamos ver que ele vai retornar para nós um Empty set, que significa que está vazio, que não tem nada, nós acabamos de criar essa tabela, de fato não teria produto.

[09:18] Então vamos inserir um produto nessa tabela, que vai ter o nome, a descrição e os valores serão: quero um notebook, um notebook Samsung. `INSERT INTO PRODUTO (nome, descricao) VALUES ('NOTEBOOK', 'NOTEBOOK SAMSUNG');` . Está errado

aqui, `descricao`. O que eu? Na verdade, eu acho que eu errei - é verdade, eu escrevi errado na hora de criar a tabela e agora ele está errado.

[10:02] Mas isso aqui nós corrigimos, é só um só um nome mesmo, isso depois fazemos um `ALTER TABLE`, que nas próximas aulas vamos conseguir ver isso melhor. Mas agora, se vocês forem reparar aqui, se eu fizer o mesmo comando de antes, `SELECT * FROM PRODUTO`, nós temos já o nosso primeiro produto.

[10:23] É importante, só mostrar para vocês aqui, com Id já inserido, ou seja, o nosso Id foi inserido sem precisarmos controlar ele, sem precisar nós mesmos, manualmente, colocar o Id. Nós vimos agora que o nosso banco de dados está funcionando, nós inserimos um produto na nossa tabela de produtos, na nossa Database que foi criada.

[10:51] Então, agora, o próximo passo é fazer com que uma aplicação em Java converse com o nosso banco de dados. Então, para a próxima aula, eu espero o ambiente de vocês configurado. E aqui também é importante mostrar para vocês que eu vou utilizar o Eclipse Jee 2019-6, mas você pode utilizar qualquer outra versão, tem a June, tem versões mais antigas no Eclipse, que não vai ter diferença no nosso curso se você estiver com ela.

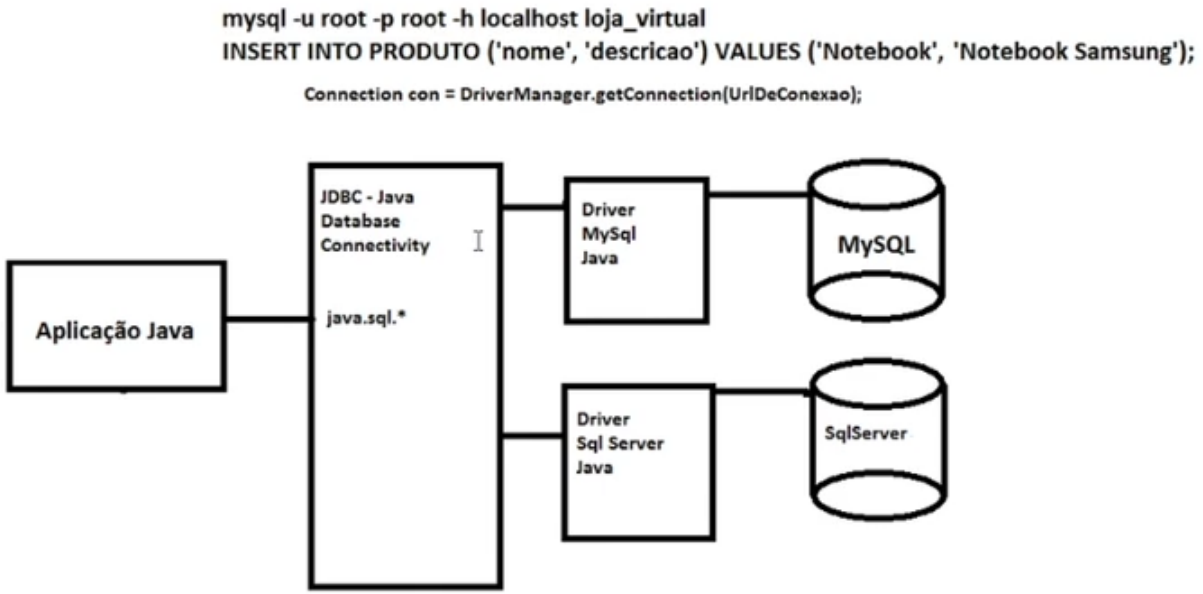
[11:42] O importante é você ter uma IDE - e pode ser outra também, não só Eclipse, para que nós possamos ter uma facilidade na hora de escrever os nossos códigos. Uma outra coisa também, que é bom vocês verificarem, é a versão do Java. No meu caso, eu estou na 11, mas nada impede de vocês utilizarem outras versões.

[12:11] Acredito que da versão 6 para frente não vai ter nenhum problema de compatibilidade. Claro que quanto mais nova a versão, às vezes pode ter certas facilidades na hora de trabalhar com o código, mas acredito que nós não vamos entrar nesse nível de código. Então da versão 6 em diante está tudo certo. Então é isso aluno, espero que você tenha gostado e até o próximo vídeo.

Transcrição

No desenho, no começo do vídeo, é mostrado o comando necessário para inserir um produto: `INSERT INTO PRODUTO ('nome', 'descricao') VALUES ('Notebook', 'Notebook Samsung');` Esse comando não irá funcionar caso seja executado, pois não há aspas simples (') nos atributos da tabela. O correto seria: `INSERT INTO PRODUTO (nome, descricao) VALUES ('Notebook', 'Notebook Samsung');`

[00:00] Fala, aluno, tudo bom? Na última aula, nós aprendemos como configurar o nosso banco de dados. Uma vez configurado, nós conseguimos recuperar a conexão, nós conseguimos nos conectar no banco de dados e realizar alguns comandos. No nosso caso, nós inserimos um produto na nossa tabela Produto.



[00:22] A nossa motivação agora é que nós consigamos fazer essa mesma coisa, recuperar uma conexão, ou inserir um produto, ou listar um produto, mas agora a partir da nossa aplicação Java. A grande questão aqui é que a nossa aplicação, ela não consegue conversar com o nosso banco de dados de forma nativa.

[00:46] Aqui nós temos uma aplicação Java rodando Java, e aqui, do outro lado, nós temos um banco de dados que roda um protocolo conhecido só por ele. Então o Java não tem como chegar no banco de dados e simplesmente fazer uma conexão com o MySQL, por exemplo. Então, como nós vamos conseguir fazer essa conexão?

[01:09] Para facilitar a vida do desenvolvedor, os desenvolvedores, a equipe do MySQL, ela criou para nós uma biblioteca Java. Uma biblioteca Java aqui, entre a aplicação e o banco de dados. A biblioteca vai conhecer todo esse lado do MySQL e vai expor para a nossa aplicação, então ela vai expor de uma forma que a nossa aplicação conheça tudo o que eu preciso para me comunicar com o MySQL.

[01:52] Então essa biblioteca vai ser o nosso famoso JAR, o JAR que nós já conhecemos da nossa linguagem Java. Então esse JAR, ele é desenvolvido pela equipe de desenvolvimento do MySQL. Quando a minha aplicação chega no JAR, eu tenho alguns conjuntos no JAR de classes, de interfaces, que dado determinado comando, vai conseguir chegar no meu banco de dados.

[02:27] E essa biblioteca aqui, ela tem um nome, ela é na verdade um driver. Então para ficar mais bonito o desenho, vamos colocar o nome que é dado para essa biblioteca, esse JAR. Então ela é um driver. Então tenho aqui o meu driver MySQL Java. Agora, com esse driver, se eu quiser me comunicar, por exemplo, com outro banco de dados, que eu vou pegar, por exemplo, um SQL Server.

[03:10] Então agora eu não tenho mais um MySQL, eu tenho um SQL Server. Eu vou precisar só pegar um driver SQL, que a nossa aplicação vai conseguir se comunicar com o banco de dados SQL Server. Então seria algo semelhante a um driver SQL Server do Java. Uma vez que tenho esse novo driver, agora a minha aplicação passa a conhecer o protocolo que é utilizado no SQL Server. Então ligar aqui para o nosso desenho ficar bonito.

[04:04] Só que nesse segundo banco, já começamos a perceber um problema. Porque, o que acontece? No primeiro driver eu tenho um conjunto de classes e interfaces do MySQL. No segundo driver, eu tenho um conjunto de classe e interfaces do SQL Server. Então dificilmente nós vamos ter chamadas iguais, vamos ter classes iguais.

[04:28] Então, para representar o que eu estou querendo falar, vamos supor que eu quero pegar uma conexão do MySQL e na classe `MySqlConnector`, por exemplo, eu tenho um `MySqlConnector.getConnection();`, que vai receber alguns parâmetros, como `(usuário, )`, que nós passamos para o nosso banco de dados, `MySqlConnector.getConnection(usuario, senha, db, servidor);` e etc.

[05:05] Muito dificilmente vamos ter algo igual do lado do SQL Server. Então, por exemplo, vamos supor que o pessoal do SQL Server criou aqui uma classe `SqlServerConnectionProvider` e tem um método `SqlServerConnectionProvider.connect()`, que também pode receber um `SqlServerConnectionProvider.connect(usuário, senha);`, enfim, pode receber os atributos que precisam de conexão com o SQL Server.

[05:52] Então aqui nós já vemos um problema: se eu precisar mudar do MySQL para o SQL Server, eu vou ter que, na hora de pegar a conexão, vou ter que alterar a chamada desse método para eu poder utilizar um outro tipo de banco. Aqui nós estamos trabalhando com duas opções, mas ainda temos bancos como PostgreSQL e qualquer um que quiséssemos nos conectar dessa maneira, seria bem trabalhoso.

[06:28] Então foi aí que o Java nos facilitou com uma camada de abstração, que vai ficar antes desses drivers. Então aqui, para ajeitar o nosso desenho, vamos mudar a forma como estávamos pensando aqui. E eu tenho aqui, após a aplicação Java, essa abstração, que vai ficar entre os drivers e vai ficar entre a minha aplicação.

[07:14] Essa abstração, ela é chamada de JDBC, ou melhor dizendo, Java Database - calma, ele não pulou linha, ficou ruim de enxergar. Então vai ser Java Database Connectivity. Agora, esse JDBC também vai ter uma abstração, ou seja, essas bibliotecas, os drivers aqui, deverão implementar os métodos que eu tenho no meu JDBC. Agora a minha aplicação só precisa conhecer esse JDBC.

[08:00] Então com a minha aplicação conhecendo JDBC, o JDBC dado algum comando, ele vai saber para qual aplicação ou para qual banco de dados que eu quero me conectar. Esse JDBC nada mais é do que o nosso pacote java.sql. Então tudo o que tem dentro de java.sql, é o nosso JDBC, essa nossa camada, essa nossa abstração da conexão com o banco de dados.

[08:43] E para conseguirmos pegar uma conexão com qualquer banco de dados que quisermos nos comunicar, eu vou ter no JDBC uma interface chamada "Connection com" e eu tenho no java.sql um `Connection con = DriverManager();` , que ele vai pegar uma conexão. Como ele vai saber qual é a conexão que é para recuperar? Dentro dos parênteses, eu vou ter uma `Connection con = DriverManager(UrlDeConexao); .`

[09:22] Com essa URL, eu vou passar qual é o tipo do banco de dados que eu quero me comunicar, eu vou falar qual é o meu usuário, qual é a minha senha, vou falar onde que está esse banco de dados e qual é a minha Database. Uma vez que eu tenho aqui um *connection*, eu tenho a minha conexão recuperada, consigo agora, da minha aplicação, realizar qualquer comando que eu quiser.

[09:55] Então nesse desenho, nós já conseguimos ver que dessa maneira, pouco importa qual é o tipo de banco que eu vou ter do outro lado. Com a minha camada de abstração, com as minhas interfaces e com esse conjunto de classes que o JDBC traz para nós, essa conexão, para nós, fica bem mais simples, fica bem mais fácil de se conectar a um banco de dados. Então é isso, aluno. Espero que tenham gostado e até a próxima aula.

Transcrição

Você pode baixar o driver do MySQL [aqui (https://caelum-online-public.s3.amazonaws.com/1451-jdbc/01/mysql-connector-java-8.0.17.jar)](https://caelum-online-public.s3.amazonaws.com/1451-jdbc/01/mysql-connector-java-8.0.17.jar).

[00:00] Bom, aluno, tudo bom? Agora que nós temos um entendimento de como a nossa aplicação Java comunica com o banco de dados, é hora de botarmos a mão na massa. Eu criei no Eclipse um projeto chamado "loja-virtual-repository" e nele nós vamos clicar com o botão direito do mouse, selecionar "New > Project" e criar uma classe chamada "TestaConexão" no campo "Name:" da janela "New Java Class". Nessa classe, eu vou querer que tenha o método `main`.

[00:30] Então uma vez que eu mande criar, nós temos uma estrutura de classe bem simples, nada que já não conhecemos. A primeira coisa que eu quero aqui é recuperar a conexão. Então eu pego a `Connection connection =`, eu pego a interface *connection*, que está dentro de java.sql. Vou pegar o `Connection connection = DriverManager`, o nosso gerenciador de drivers.

[01:00] Dentro do gerenciador eu pego o método `Connection connection = DriverManager.getConnection(url, user, password);`, que possui a string URL, a string user e a string password. Aqui, nesse arquivo texto, eu já separei a nossa string de conexão com o banco de dados que nós configuramos nas aulas anteriores. Então a nossa string, aqui entre parênteses, ela vai ser bem simples.

[01:25] E aqui, a string é bem ilegível mesmo, nós temos `jdbc:`, da especificação JDBC, o banco de dados, qual é o banco de dados, então no nosso caso nós configuramos o MySQL. Vai estar na nossa máquina mesmo, o `localhost`, a `loja_virtual`

foi a Database que nós criamos e tenho que fazer essas configurações de `Timezone` e `serverTimezone`. E usuário e senha, `"root", "root"`, que já tínhamos configurado anteriormente.

[01:56] Aqui, esse alerta que o Eclipse está dando, é porque quando você vai usar a aplicação Java para se comunicar com o banco de dados, nós temos N maneiras de ter uma exceção. Se o banco de dados, que eu quero me conectar, estiver fora, eu vou ter uma exceção.

[02:17] Se essa minha string de conexão estiver errada, eu vou ter uma exceção. Então o Eclipse só pede para adicionarmos um `throws SQLException`. O que eu quero fazer é só recuperar a conexão e depois fechar essa conexão, porque nós vamos ver, ao longo do curso, que tudo que nós abrimos, nós temos que fechar, então com a conexão não é diferente.

[02:45] Então a conexão que eu recuperei, se estiver tudo certo, se eu não tiver nenhuma exceção na execução, eu fecho essa conexão com `connection.close();`. O meu código, ele fica bem sucinto aqui, bem simples. E na hora de testar, é para ele só terminar o processo: ele vai executar, fecha a conexão e termina o processo do programa que estamos executando.

[03:11] Tivemos um erro aqui. Olha só, o que ele está falando, que não foi encontrado o driver. E é de fato que nós falamos anteriormente, para o Java se comunicar com o MySQL, com o SQL Server, ele precisa de um driver que conheça tudo o que tem no SQL, tudo o que tem no banco de dados. A nossa aplicação, ela não vai conseguir se comunicar de forma nativa, então precisamos desse driver.

[03:45] E, como foi falado, ele é uma lib, então ele vai ficar dentro do Build Path, eu tenho um "Build Path > Add External Archives...". Então, quando eu for selecionar, e aqui eu peço para vocês usarem a versão do arquivo que está no repositório da Alura, que é o "mysql-connector-java-8.0.17".

[04:08] Vamos trabalhar sempre na mesma versão para evitar qualquer tipo de incompatibilidade de versões mais novas, versões mais antigas. Pode ser que lá na frente nós não consigamos executar os mesmos comandos por conta de uma versão, então, para não ter nenhum tipo de problema, use esta versão que está no repositório.

[04:29] Agora vou mandar executar de novo o código. Uma vez que eu mando, beleza, não tivemos nenhuma exceção, o processo aqui, o nosso programa rodou com sucesso. Para ficar melhor de visualizar essa execução, vou botar aqui `System.out.println("Fechando conexão! !");` depois que ele recuperou a conexão.

[04:52] Então se eu vir em "Run as > Java Application" e mandar executar de novo, vocês viram, ele demorou um pouco aqui, recuperou a conexão, fechou a conexão e depois deu um *close*. Por que sabemos que deu tudo certo? Porque nós não tivemos nenhuma exceção, nenhum SQL *exception*.

[05:12] Então é isso, aluno. Agora nós estamos caminhando já para partes mais avançadas dessa comunicação da nossa aplicação com o banco de dados. Agora com essa conexão em mãos, nós vamos conseguir executar aqueles comandos SQL, como Insert, como Update, como Select, tudo isso que nós estamos ansiosos já para ver. Então espero que tenham gostado e até a próxima aula.

O que aprendemos?

Nesta aula, aprendemos que:

- Para acessar o banco de dados, precisamos de um ***driver***
  - Um *driver* nada mais é do que uma biblioteca (JAR)
- **JDBC** significa *Java DataBase Conectivity*
  - JDBC define uma camada de abstração entre a sua aplicação e o *driver* do banco de dados
  - Essa camada possui, na sua grande maioria, interfaces que o *driver* implementa
- Para abrir uma conexão, devemos usar o método `getConnection`, da classe `DriverManager`
  - O método `getConnection` recebe uma string de conexão JDBC, que define a URL, usuário, senha, etc

Transcrição

[00:00] Olá, aluno. Tudo bom? Anteriormente nós vimos como que nós vamos fazer para pegar uma conexão do nosso banco de dados a partir da nossa aplicação. Para isso nós usamos uma classe de teste, chamada `TestaConexao`, onde nós passamos a nossa string de conexão com as informações para a nossa aplicação achar o banco de dados.

[00:20] E apenas fechamos a conexão, porque nós vimos que os recursos do banco de dados, quando acessados pela aplicação, nós devemos fechá-los. Bom, esse código funcionou, se executarmos vamos ver que a nossa conexão está abrindo normalmente, só que a nossa ideia de abrir uma conexão com o banco de dados é para podermos fazer aqui as operações com banco de dados, Inserts, os Selects, os Updates.

[00:49] Enfim, tudo o que nós executamos no banco de dados nós temos que fazer aqui, a partir da nossa aplicação. E eu quero começar aqui trazendo as informações que nós gravamos no banco de dados, lá quando estávamos configurando a nossa Database. Então se viermos no MySQL 8.0 Command Line Client, vou botar a minha senha aqui.

[01:11] Eu vou usar aqui a nossa Database que nós criamos, que é a nossa `use loja_virtual`. Se eu fizer um `select * from produto;`, nós vamos ver que nós temos dois produtos. E é esse produto que eu quero trazer na minha aplicação. Então como nós vamos fazer? Aqui no Eclipse, dentro do nosso projeto, em "loja-virtual-repository > src > (default package)", à esquerda, eu vou criar uma nova classe, que vai se chamar "TestaListagem".

[01:48] Essa classe vai ter um `public static void main(String[] args)` e nós vamos copiar esse conteúdo da classe "TestaConexão". Nós vamos precisar tanto da string de conexão quanto fechar a nossa conexão, então só preciso dar um "Ctrl

+ C" no código da classe "TestaConexao" e um "Ctrl + V" no código da nova classe. Agora temos o necessário para pegar essa conexão com o banco de dados.

[02:10] Vou adicionar o `throws SQLException`, que nós vimos que é necessário, porque o SQL, ele pode, nessa comunicação com o banco de dados, ele pode dar o SQL Exception, então nós temos que avisar para quem for chamar esses métodos aqui. Então no nosso caso, o main, ele tem que avisar aqui, esse código pode dar um SQL Exception. Como que nós fazemos então para usar os comandos de banco de dados na nossa aplicação?

[02:36] Aquela cláusula que nós usamos no nosso banco de dados, no MYSQL, o `select * from`, o `select` e outros campos que nós queremos trazer para dentro da nossa tabela, eles são considerados no mundo Java como Statements. E como eu faço para criar um Statement? Quando recuperamos a conexão, eu vou ter um método chamado `con.createStatement();`.

[03:05] Esse comando, ele me devolve um Statement. Nós vamos ver aqui, eu tenho uma interface, cuidado para não confundir, que eu tenho uma `Statement` do MySQL, eu quero a "Statement - java.sql", que é o nosso pacote JDBC. Esse `Statement stm = con.createStatement();` agora me devolve um Statement. Como que eu faço então para criar a minha cláusula SQL, o meu Statement SQL?

[03:35] Eu tenho aqui o meu Statement, a minha referência, e eu vou chamar o método `stm.execute();`. Então o que eu passo aqui, entre parênteses? A cláusula que eu quero, que é um select, eu quero trazer o ID do meu produto, o nome do meu produto e a descrição do meu produto. E eu apontei que é da minha tabela produto. Então fica `stm.execute("SELECT ID, NOME, DESCRICAO FROM PRODUTO");`.

[03:57] Então com essa linha nós fizemos mais ou menos o comando que nós executávamos no banco de dados, mas agora nós estamos trabalhando na nossa aplicação. Perfeito? Nós vimos que eu tenho 1 ou N produtos na minha tabela, eu posso ter vários produtos, conforme eu vou adicionando. Então, provavelmente, essa Query `SELECT`, esse nosso Statement vai nos devolver uma lista.

[04:21] Então vamos ver se ele devolve uma lista. Será que é uma lista de string? Vou fazer aqui então, vamos ver se é uma lista de string. Vou botar aqui `List<String> resultados = stm.execute("SELECT ID, NOME, DESCRICAO FROM PRODUTO");`. E se eu fizer esse comando, ele vai dar uma falha na compilação e quando nós vamos ver, ele vai falar que não pode converter um tipo booleano para um tipo string.

[04:42] Então significa que essa linha nos devolve um booleano? Estranho, mas vamos mudar então para o tipo boolean, para vermos o porquê que esse comando nos devolve um booleano. Então fica `boolean resultados = stm.execute("SELECT ID, NOME, DESCRICAO FROM PRODUTO");`. Aparentemente compilou agora. E qual é a motivação desse código, que nos devolveu um booleano?

[05:04] Bom, quando vamos trabalhar com banco de dados, nós temos aquelas cláusulas padrões que nós utilizamos, as mais conhecidas, que é o select, o insert, o update, o delete. E o que acontece? O `.execute`, ele vai nos retornar um booleano true quando o retorno do meu Statement for uma lista.

[05:26] Ou seja, quando eu fizer uma operação com o banco de dados, a partir da minha aplicação, e esse resultado for a lista, que no nosso caso é um select. E quando o retorno for diferente de uma lista, por exemplo, um delete, que não me retorna nada, um insert, que não me retorna nada, um update, que não me retorna nada, esse booleano vai ser *false*.

[05:46] Então, para testarmos isso, eu posso dar um `System.out.println(resultado);` e nós vamos ver se realmente ele nos retorna um true. Se eu mandar executar esse código, vou clicar com o botão direto, dar um "Run as > Java Application" e está aqui o resultado true, porque de fato ele é uma lista. Só que para eu pegar essa lista, eu não pego diretamente aqui igual nós queríamos, uma lista de string. Como eu tenho que fazer?

[06:12] Eu vou tirar essa variável resultado daqui, `boolean resultados = stm.execute("SELECT ID, NOME, DESCRICAO FROM PRODUTO");` e nós vamos utilizar o `Statement` para pegar os resultados dessa lista. Para eu pegar os resultados dessa lista com `Statement`, eu tenho um `getResultSet();`. Com esse `stm.getResultSet();`, eu tenho também uma interface chamada `ResultSet`, também do pacote java.sql, do nosso JDBC.

[06:44] E com `ResultSet rst stm.getResultSet();` eu consigo pegar todo o conteúdo dos meus produtos adicionados à minha tabela. Então você pode ver que agora está compilando. E como que eu faço então para pegar os meus resultados da minha tabela? Eu tenho que verificar se quando eu for buscar, na minha tabela, se eu tenho um próximo. Então como que eu faço?

[07:11] Eu quero pegar o primeiro produto. Beleza, eu pego o primeiro produto. Eu tenho um próximo? Tenho. Então vai buscar o próximo produto. Eu tenho um próximo? Então eu busco o próximo produto. Quando não tiver mais produtos, ele sai do laço. Então a palavra-chave aqui é um laço e se eu tenho próximo. Eu posso usar como laço aqui um `while`.

[07:33] E, para facilitar a nossa vida, o próximo que sempre vamos procurar o `ResultSet`, ele já nos fornece com esse método `while(rst.next()) {}`. Então ele vai pegar na primeira posição. Tem um próximo? Para ficar mais claro, vamos no MySQL 8.0 Command Line Client. O ID vai estar como se fosse uma posição zero aqui na tabela. Quando eu chamo o `(srt.next());`, ele vai perguntar: tem um próximo cara que vai ser o primeiro produto? Tem.

[08:01] Então pega esse cara, o primeiro produto. Tem um próximo cara? Tem. Então pego esse segundo produto. Tem um próximo? Não. Então eu saio do laço. Então, dessa forma, nós conseguimos trazer os dois produtos. Mas ainda não terminamos, precisamos pegar os atributos, nós precisamos montar aqui, na verdade, os atributos no mundo Java com as informações do MySQL que nós temos na tabela produto. Então, como eu faço isso?

[08:26] Nosso primeiro atributo é o ID, então eu pego ele como `integer id`. E o `rst`, o ResultSet, vamos ter um `srt.getInt` está vendo aqui? E nós temos duas formas de buscar esse `.getInt`, que é de fato, a coluna que queremos pegar. Então eu quero pegar o resultado da coluna ID, eu tenho duas formas para fazer isso: informando com `(int columnIndex)`, que no MySQL vai ser o primeiro Index ou com o `(String columnLabel)`.

[09:09] Para ficar claro, o que acontece? Eu vou pegar o `(String columnLabel)` e vou passar o ID. Se eu quisesse pegar pelo `(int columnIndex)`, o Index do nosso ID é o 1. Lembrando que é diferente de uma lista do Java, por exemplo, que começa no Index 0, aqui vai começar do Index 1. Então tenho o ID 1, o Index 1, o nome o Index 2 e a descrição o Index 3.

[09:36] Se eu quiser pegar pelo Label, eu vou usar ID, nome e descrição, e eu posso utilizar ele como ID maiúsculo que vai funcionar. Se eu fizer um `Integer id = rst.getInt("ID");` e `System.out.println(id);`, nós vamos ver o ID. Pra eu pegar o nome, o nosso conteúdo vai ser uma string, então eu uso o `String nome = rst.getString();`. Então você vê que é bem fácil de trabalhar com a recuperação dos dados da nossa tabela a partir da nossa aplicação.

[10:04] E eu informo aqui o nome, o `column label` da nossa tabela. Então fica `String nome = rst.getString("NOME");`. E depois também dou um `System.out.println(nome);` passando o nome. E, por último, que nós precisamos, é a nossa descrição, que também vai ser um `rst.getString()`, e vamos passar ele como `String.descricao = rst.getString("DESCRICAO");`.

[10:33] Se eu der um `System.out.println(descricao);`, nós vamos pegar a descrição e vamos então printar na nossa tela. Então, dessa forma você vê que o método está bem explicado. Eu executo a Query em `stm.execute("SELECT ID, NOME, DESCRICAO FROM PRODUTO");`, pego o resultado em `ResultSet rst = stm.getResultSet();`.

[10:48] Faço o laço nesse resultado, em `while(rst.netx()){` e imprimo o ID, o nome e a descrição do meu produto, que está na tabela. Se eu mandar executar esse laço, vou dar um "Run as > Java Application", ele vai nos trazer os dois produtos que estavam na nossa tabela. Então agora já temos o nosso primeiro método de operação, fazendo mesmo a operação com o nosso banco de dados a partir da nossa aplicação.

[11:14] O intuito agora é que continuemos com esse trabalho, continuemos realizando as outras operações, como delete, insert, entre outras que nós vamos ver ao longo do curso. Mas nessa aula, a nossa intenção era criar aqui a listagem. Dou a aula como concluída e eu vejo você no próximo vídeo. Obrigado, aluno.

Transcrição

[00:00] Fala, aluno. Tudo bom? Nas últimas aulas nós vimos como recuperar uma conexão no nosso banco de dados e como listar os produtos que foram inseridos na nossa tabela. Aparentemente, o código, ele está ok. Se eu mandar executar a nossa classe `TestaConexao`, ele vai retornar o que nós esperamos, que é uma string informando que a conexão foi fechada. E a `TestaListagem` vai listar todos os produtos que temos na nossa tabela.

[00:36] Se analisarmos bem os dois códigos, nós temos aqui, em `System.out.println("Fechando Conexão! !");` um ponto de atenção. Quando eu recupero a conexão, eu estou utilizando todo esse trecho de código aqui, do `Connection connection = DriverManager`, chamando o Driver manager, o gerenciador de driver, chamando o método `.getConnection` e ainda passando a string de conexão. Na `TestaListagem`, a mesma coisa.

[01:04] Qual é o ponto de atenção aqui? Se um dia eu mudar a minha senha do banco de dados, de root para 123, eu vou ter que mudar nessa classe `TestaListagem` e nessa classe `TestaConexao`. Se eu tiver uma classe Testa Inserção, eu vou ter que trazer todo esse código da `TestaConexao` para a Testa Inserção. Então nós vemos que vai aumentando a complexidade, aumentando o risco de quebra do nosso código.

[01:30] Então, para isso, para refatorarmos o nosso código, eu vou criar uma nova classe chamada `CriaConexao`. Dentro dessa classe `public Class CriaConexao` eu vou ter um método, que vai retornar uma Connection e vou botar aqui `public Connection recuperarConexao`. O nome fica a critério de vocês, o que vocês acharem melhor. O meu aqui vai ser `recuperarConexao`.

[02:01] E toda essa linha de código aqui, a `Connection connection = DriverManager` em `TestaListagem`, eu posso retirar e voltar as linhas para `CriarConexao`. Eu vou precisar adicionar o `throws SQLException` no `CriaConexao` e vou ter que colocar um `return DriverManager`, que ele retorna uma Connection.

[02:20] O código, ele vai ficar aqui, nessa minha `CriaConexao`. E, na `TestaListagem`, eu não preciso mais passar todo aquele código, eu sou vou precisar instanciar aqui a minha classe, que eu acabei de criar: `CriarConexao criaConexao = new CriaConexao();`. Vou receber uma `Connection connection = criaConexao.recuperarConexao();` do método `.recuperarConexao();`.

[02:53] Já demos uma melhorada no código aqui. Vou remover o `import java.sql.DriverManager` que não está sendo mais utilizado, que é do Driver manager. A mesma coisa eu vou fazer na minha `TestaConexao`. Vou apagar o código, eu instancio a classe que acabamos de criar, a `CriaConexao criaConexao = new CriaConexao();`.

[03:16] E chamo o método `.recuperarConexao();`, que vai devolver uma `Connection connection = criaConexao.recuperarConexao();`. Vamos novamente retirar esse `import java.sql.DriverManager`, que não está sendo mais utilizado. Se eu mandar executar, o retorno do código vai ser a mesma coisa que tínhamos visto no começo da aula. Ele vai informar, é a mesma string `("Fechando Conexão! !")`.

[03:47] Se eu executar a `TestaListagem`, nós vamos ver a mesma lista que nós tínhamos visto anteriormente. Então vemos que agora o código, ele ficou encapsulado, ele ficou centralizado em uma única classe, a `CriarConexao`. Se eu precisar mudar alguma coisa na minha string de conexão, se eu precisar mudar o meu banco de dados de MySQL para SQL, eu só vou precisar mexer nessa nossa classe `CriaConexao`.

[04:18] Só que, o mais interessante disso tudo, é que esse conceito ele não foi criado por mim agora. Nós temos, no Java, um Design pattern, ou seja, um padrão de projeto, que se chama Factory Method. Esse Factory Method tem por objetivo centralizar, encapsular um código que vai criar um objeto. Ou seja, essa classe `CriaConexao`, ela vai ser uma fábrica de conexões.

[04:52] Então toda vez que essa classe for chamada, é porque eu quero uma conexão. Então, esse Factory Method, ele tem vários outros conceitos aplicados nele, vocês podem pesquisar que é bem interessante. Para ficar mais claro aqui, eu vou refatorar o nome da minha classe clicando sobre "CriaConexao.java" com o botão direito para selecionar "Refactor > Rename..." e vou botar como uma "ConnectionFactory" no campo "New Name" da janela "Rename Compilation Unit", porque é isso o que ela é.

[05:25] Ela é uma fábrica de conexões, sempre que eu precisar de uma conexão, eu vou chamar a minha `ConnectionFactory`. Com isso, agora o nosso código fica bem bacana, porque além de nós centralizarmos o nosso código, encapsulamos essa instanciação de uma conexão, nós já estamos utilizando um padrão que é utilizado pelos desenvolvedores da linguagem Java.

[06:04] Então, só para ficar mais bonito aqui, eu vou mudar o nome da minha variável de `CriaConexao` para `ConnectionFactory` no restante do código, e agora nós temos a nossa `ConnectionFactory` funcionando normalmente. Só para testar, vou executar mais uma vez as duas classes e nós vamos ver que o resultado foi o mesmo. Então é isso, aluno. Espero que vocês tenham gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Vamos dar continuidade ao nosso curso de JDBC. Recapitulando as aulas anteriores, principalmente a nossa classe `TestaListagem`, se nós executarmos ela, nós vamos ver que tem os dois produtos inseridos, um notebook e a geladeira. Esses produtos, eles foram inseridos na época em que estávamos ainda configurando o nosso banco de dados.

[00:24] Então, eles foram inseridos via banco de dados e não via aplicação. A intenção agora é que nós consigamos fazer essa inserção pela a nossa aplicação. Para isso, nós vamos criar uma nova classe chamada "TestaInsercao". Essa classe, ela vai ter um método `public static void main(String[] args)`.

[00:49] E vamos utilizar a nossa `ConnectionFactory factory = new ConnectionFactory();` para poder usar o método `recuperarConexao`, que vai ser o que de fato vai retornar a conexão para a nossa classe `TestaInsercao`. Então vamos `Connection connection = factory.recuperarConexao();`.

[01:11] Aqui, como nós não estamos tratando em `.recuperarConexao` a SQL Exception, então nós temos que adicionar o `throws SQLException` no nosso método `main`. Beleza, uma vez que eu tenho a conexão em mãos agora, nós vimos que para executar cláusulas SQL a partir da nossa aplicação, nós precisamos de um Statement.

[01:34] Esse Statement, para recuperarmos ele, nós precisamos criar o Statement aqui, que está dentro da connection. Nós temos o `Statement stm connection.createStatement();`, que vai nos retornar o Statement. Tenho agora o `Statement stm connection.createStatement();` em mãos, agora eu posso chamar o método `stm.execute("");` e passar para ele qual é a cláusula que nós queremos.

[02:01] Se nós queremos inserir algo, vai ser um `stm.execute("INSERT INTO PRODUTO ()");`. Então vamos inserir no produto um nome e uma descrição. Os valores, eu quero aqui um mouse e a descrição dele vai ser um mouse sem fio. Fica então `stm.execute("INSERT INTO PRODUTO (nome, descricao) VALUES ('Mouse', 'Mouse sem fio')");`.

[02:24] Um ponto de atenção aqui é nas aspas. Lembrando que aspa simples, aqui, nesse caso, é para a string do SQL, então da cláusula do SQL, e as aspas duplas vão ser para a string do Java. Se inverter as aspas vai dar erro, então só para evitar esse tipo de problema mesmo.

[02:48] Nós vimos, anteriormente, que o método `.execute`, ele nos retorna um booleano. Quando ele retorna uma lista, o booleano vai ser true, quando o retorno da cláusula for uma lista, o booleano vai ser true, quando não, vai ser false. No nosso caso, é um insert, o insert não vai retornar uma lista, então ele tem que ser false.

[03:12] Para comprovamos isso que estamos falando, basta fazermos um `System.out.println(resultado);` e ele vai ter que ser false. Beleza. Mas para nós não é um resultado muito agradável. Como que eu vou utilizar isso na minha aplicação? De que me serve esse false? Não faz sentido.

[03:40] Para mim, faria mais sentido eu ter um resultado, de quando inserisse um produto, de qual foi o produto que foi criado. Qual foi o ID desse produto? Então, para isso, aqui nas nossas classes e métodos JDBC, nós temos uma riqueza de recursos para trabalhar com o banco.

[04:07] Dentro do `stm.execute`, ele me permite criar um novo parâmetro, que vai ser o Statement e dentro dele tem `RETURN_GENERATED_KEYS`, então fica: `stm.execute("INSERT INTO PRODUTO (nome, descricao) VALUES ('Mouse', 'Mouse sem fio')", Statement.RETURN_GENERATED_KEYS);`.

[04:20] Então estou falando para ele o seguinte: quando eu executar essa cláusula insert, eu quero também que ele me retorne a chave gerada, o ID gerado. Para nós, isso vai ser interessante nesse seguinte ponto, eu executo, depois eu pego a chave gerada dentro do meu Statement, com `stm.getGeneratedKeys();`.

[04:47] E esse `getGeneratedKeys();`, ele vai me retornar um `ResultSet rst = stm.getGeneratedKeys();` que nós conhecemos já, que nós sabemos iterar nesse `ResultSet`. Aqui eu vou fazer o `while`, que nós já vimos anteriormente, `while(rst.next())` e aqui - não sei se vocês lembram. Quer dizer, espero que vocês lembrem.

[05:09] Quando nós vamos buscar informação de uma coluna, nós estamos buscando pelo Label dessa coluna. Para ficar mais fácil de relembrar, caso alguém tenha esquecido, dentro da coluna nome, eu passo a Label `("NOME")` mesmo para recuperar o valor que vai estar naquela coluna.

[05:34] Nós temos uma maneira diferente para fazer isso aqui, e para vocês terem conhecimento, no nosso caso nós queremos saber qual foi o ID criado, então eu vou pegar `Integer id = rst.getInt`, igual anteriormente, só que invés de eu fazer o "String columnLabel", eu vou fazer o "Int columnIndex".

[05:57] E aqui eu vou passar o index 1, com `Integer id = rst.getInt(1);`, que no SQL a primeira coluna, ela é considerada 1, não é 0 igual em uma lista Java, por exemplo. E para eu verificar aqui agora qual é o resultado disso, eu vou botar a seguinte mensagem: `System.out.println("O id criado foi: " + id);` e vou pegar o ID, que nós estamos recuperando.

[06:30] A partir do momento que eu dou um "Run as", ele vai me mostrar o ID que foi criado, que foi o 18. Se eu for na minha `TestaListagem`, agora eu vou verificar que ele criou para nós o produto mouse sem fio com o ID 18. Se eu executar mais uma vez o código `TestaInsercao`, o ID criado foi 19. E se eu verificar em `TestaListagem`, ele vai mostrar os dois mouses.

[07:09] Então a cada vez que eu testar a minha classe `TestaInsercao`, ele vai criar um novo produto. Só que nesse caso, nós verificamos que ele criou alguns lixos. Nós testamos a `TestaInsercao`, vimos que está inserindo, muito bom, por sinal o nosso resultado. Porém, agora nós temos lixo.

[07:33] Para lixo, nós temos que apagar esses registros repetidos, nós não queremos uma base com registros repetidos, não faz sentido, a nossa tabela não vai ficar concisa. Então isso é um desafio para as próximas aulas, para nós apagarmos esse lixo. Mas, por enquanto, nós temos uma nova classe inserindo um novo produto, retornando o ID desse produto e é isso. Espero que vocês tenham gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Anteriormente, nós vimos como inserir um produto a partir da nossa aplicação. Nós criamos a classe `TestaInsercao`, criamos uma cláusula insert para passar o mouse como produto que nós queremos agora na nossa base de dados. Porém, efetuando alguns testes, esse produto novo, ele foi inserido várias vezes. Se nós bem lembramos, a nossa tabela ficou com informações repetidas.

[00:36] Sendo que não tem necessidade, eu não quero informações repetidas na minha tabela. Eu quero apagar esses dados. Como agora tudo o que nós fazemos que envolva o banco de dados, é a partir da nossa aplicação, para remover esses dados repetidos também eu vou utilizar a aplicação.

[00:54] Então, para isso, eu vou uma classe "TestaRemocao". E eu quero deixar só os produtos que nós inserimos quando estávamos configurando o nosso banco de dados, então vai ser a geladeira azul e o notebook Samsung. Nessa nova classe, nós vamos utilizar todos os conceitos que já vimos ao longo do curso.

[01:18] Eu vou ter que instanciar uma `ConnectionFactory` - calma, sem antes criar o `main` não. Então eu crio um `public static void main(String[] args)` e agora sim eu vou instanciar a minha `ConnectionFactory`, que eu vou precisar de uma conexão aqui - `ConnectionFactory factory = new ConnectionFactory()`.

[01:43] Na minha `factory` eu vou chamar o `Connection connection = factory. recuperarConexao();`. Vou precisar adicionar o `throws SQLException`. Após eu recuperar a conexão, eu preciso criar o Statement, vou receber aqui ele em uma variável chamada `stm`, então fica `Statement stm = connection.createStatement();`.

[02:11] E vou chamar o `stm.execute();`. Como nós queremos deixar só os dois primeiros, então vou fazer a cláusula `("DELETE FROM PRODUTO")` onde eu quero apagar os IDs maior que 2, então fica `stm.execute("DELETE FROM PRODUTO WHERE ID > 2");`. Se eu executar esse código, pode ser que funcione, pode ser que não. Eu preciso de uma confirmação.

[02:40] Qual confirmação eu consigo ter após deletar algumas linhas no nosso banco de dados? Aqui, no Statement, eu tenho um método que chama `getUpdateCount()`, que nos retorna um inteiro. Esse inteiro significa o seguinte, quantas linhas que foram modificadas após o Statement ser executado. Então, quando eu executo esse delete do meu código, quantas linhas ele vai apagar?

[03:14] Isso nós podemos verificar da seguinte forma: eu vou criar uma variável inteira e vou chamar de `linhasModificadas`, ficando `Integer linhasModificadas = stm.getUpdateCount();`. E, para vermos esse resultado melhor, eu vou botar um `System.out.println("Quantidade de linhas que foram modificadas: " + linhasModificadas);`.

[03:41] Se eu executar esse trecho de código, do `TestaRemocao`, quando eu der o "Run As > Java Application", ele vai me mostrar quantas linhas foram excluídas. Então três linhas foram excluídas com essa cláusula SQL. Se de fato nós voltarmos na nossa classe `TestaListagem` e executarmos ela novamente, o resultado vai ser o que nós esperávamos, apenas os dois produtos que foram inseridos desde as primeiras aulas.

[04:11] Se eu voltar na `TestaRemocao` e manda executar o código novamente, ele vai me retornar que foram zero linhas modificadas, porque já não tem mais produtos com ID maior que 2 para serem deletados, então ele não fez nada, não teve alteração no nosso banco de dados. Então por isso esse valor de resultado é zero.

[04:32] Então, com o `TestaRemocao`, nós fizemos três comandos básicos do SQL, que foi o delete, o insert e o select. A ideia agora é que nós iremos avançar e vamos entrar em conceitos um pouco mais complexos, vamos trabalhar com cláusulas mais complexas, enfim, todo esse passo a passo para entendermos como funciona esse mundo do JDBC. Então, espero que você gostado e até a próxima aula.

O que aprendemos?

Nesta aula, aprendemos que:

- Para simplificar e encapsular a criação da conexão, devemos usar uma classe `ConnectionFactory`
  - A classe `ConnectionFactory` segue o padrão de criação *Factory Method*
  - O *Factory Method* encapsula a criação de um objeto
- Para executar um comando SQL, podemos usar a interface `java.sql.Statement`
  - O método `execute` envia o comando para o banco de dados
  - Dependendo do comando SQL, podemos recuperar a chave primária ou os registros selecionados

Transcrição

[00:00] Fala, aluno. Tudo bom? Voltando ao nosso curso de JDBC, eu gostaria de passar novamente na nossa classe `TestaInsercao`. Nessa classe, nós já realizamos alguns testes e vimos que todos foram feitos com sucesso. Porém, agora, eu queria desenvolvê-la de forma um pouco diferente.

[00:21] Vamos criar uma nova classe, chamada "TestaInsercaoComParametro", e vocês já vão entender qual a motivação dessa classe. Dentro dela nós vamos ter um método `public static void main(String[] args)` e para não perder muito tempo, vamos copiar esse mesmo código da classe `TestaInsercao` na `TestaInsercaoComParametro`.

[00:46] Vamos adicionar o `throws SQLException` na `TestaInsercaoComParametro`. Minha motivação para desenvolver essa classe é pensando o seguinte: se o sistema, um dia ele tiver uma interface gráfica e as informações como o nome e a descrição do produto forem informados através de um formulário, por exemplo, o que vai acontecer?

[01:11] Quando o usuário colocar no meu formulário o nome e a descrição e dar um submit, do meu lado da aplicação, eu tenho que recuperar essa informação, guardar os valores em uma variável e concatenar com a cláusula SQL, que aí, de fato, quando for executado o comando `stm.execute`, ele vai inserir as informações com as informações que usuário nos passou.

[01:38] Então eu vou pegar, em `TestaInsercaoComParametro`, uma variável chamada `String nome "";`, do tipo string, por enquanto eu vou iniciar essa variável com uma string vazia, descrição também do mesmo jeito: `String descricao = "";`. Agora eu só preciso concatenar essas strings na nossa cláusula insert do SQL.

[02:05] Então fica `stm.execute("INSERT INTO PRODUTO (nome, descricao) VALUES (' " + nome + " ', ' " + descricao + " '), Statement.RETURN_GENERATED_KEYS");` . Se eu inicializar com `String nome = "Mouse";` no nome e `String descricao = "Mouse sem fio";` na descrição, não é para ter nenhuma diferença da forma que estava antes.

[02:28] Nós vimos, ele criou um novo ID e na nossa classe `TestaListagem` , o produto foi criado da mesma forma. Só que agora nós estamos lidando com informações que vem de um formulário, que vem de um usuário. O usuário, ele pode ter, por exemplo, errado na hora de inserir o nome, e passou `String nome = "Mouse' ";` , com aspas simples.

[03:00] Novamente, o nome do produto está entre aspas duplas na nossa aplicação, não é para ter nenhum tipo de problema. Se eu mandar executar o `TestaInsercaoComParametro` , eu quero o ID 41. Tivemos um erro. Que erro é esse? Você tem um erro na sua sintaxe SQL. Mas o mais estranho disso é que nós não mudamos nada na nossa string.

[03:27] Na verdade o usuário errou, passou uma aspa simples junto com o nome do produto, mas era pra ter sido tratado como uma string para o SQL, era para ter salvo uma string "Mouse' ", com aspas simples. Estranho. Vamos verificar o que aconteceu. Eu vou criar uma variável chamada `sql` , em "TestaInsercaoComParametro", vai ser do tipo string também, e eu vou guardar a nossa cláusula insert do SQL nela.

[04:00] Então vai ser aqui `string sql = "INSERT INTO PRODUTO (nome, descricao) VALUES (' " + nome + " ', ' " + descricao + " ');` . Agora, invés de passar para o `stm.execute` a string, eu passo `stm.execute(sql);` , com a variável. Só que para verificar como a aplicação está montando a nossa cláusula SQL, eu vou dar um `System.out.println(sql);` na variável. O erro vai ser o mesmo, só que pelo menos agora conseguimos verificar como está a cláusula.

[04:30] Ele montou, no `VALUES` , e quando ele foi montar a cláusula, na verdade ele considerou a aspas simples, colocada por engano no nome, em vez de ser parte da string e salvar essas aspas simples como string, as aspas simples, ela é específica do SQL. Então, nos comandos SQL, quando nós vamos inserir uma string, a string ela fica, diferentemente do Java, que é entre aspas duplas, a string vai ficar entre aspas simples no SQL.

[05:08] Foi por isso que a nossa aplicação deu erro. Como as aspas simples são do domínio do SQL, dos comandos SQL, da linguagem SQL, então o erro do usuário acabou quebrando a nossa aplicação. Só que a pior parte não é o erro. Vamos supor, aqui foi um erro de usuário, o usuário é humano, suscetível ao erro.

[05:32] Mas nessa situação, nós encontramos também usuários que sabem o que estão fazendo e fazem para prejudicar o desenvolvedor, o dono do sistema, porque ele pode fazer isso, quando ele terminar de fazer a descrição que ele quer para o produto, ele pode fechar parênteses, dar um ponto e vírgula e, por exemplo, escrever uma cláusula delete - "Mouse sem fio); delete from Produto;".

[06:07] Se ele conhecer um pouco da arquitetura, se ele conhecer um pouco das tabelas, souber que tem uma tabela produto, se ele der um "delete from Produto", na hora que ele der um submmit, em vez de ele criar um novo produto, ele vai deletar toda a nossa tabela de produto. Olha só que prejuízo.

[06:28] Por causa de um simples comando que o desenvolvedor muitas vezes não está esperando, porque o usuário, ele está inserindo uma cláusula que está realizando uma ação que não é esperada. Eu espero que, nesse meu formulário, eu insira produtos e não delete os meus produtos.

[06:53] Então com o usuário um pouco mais experiente, um hacker, ou uma pessoa que está agindo de má-fé, ela consegue injetar um SQL, uma cláusula SQL e consegue quebrar a sua aplicação, te dar um prejuízo bem grande, principalmente se tratar-se de um delete. Essa ação, ela se chama exatamente como eu falei ainda agora, é uma injeção de SQL, é uma SQL Injection.

[07:26] Isso é uma prática até comum, as pessoas, elas ficam procurando vulnerabilidades nas aplicações e ficam fazendo testes para verificar, muitas vezes com a intenção de recuperar alguma informação valiosa, ou muitas vezes apenas para te dar o prejuízo mesmo, como no caso de um delete. A pessoa pode não recuperar nenhuma informação valiosa, mas também ela apaga a sua informação valiosa. Então como eu posso tratar esse problema?

[07:56] Sendo que com um pequeno erro no nome, que é uma aspa simples, eu vou quebrar a minha aplicação, e um pouco mais grave, um pouco mais grave não, bem mais grave eu deleto uma tabela toda. O que posso fazer no código, eu como desenvolvedor, é fazer uma validação caractere por caractere. Então, quando eu verificar que tem uma aspa simples, eu escapo, tudo o que for à direita, depois das aspas, eu não considero.

[08:27] Considero também que tudo o que tiver um ponto e vírgula, eu não vou considerar e não vou contar o que estiver à direita, após o ponto e vírgula. Mas olha o tanto que parece ser falho isso, porque eu sou humano. Eu, desenvolvedor, sou humano, então estou suscetível a erro também. Assim como o usuário está, ao inserir, ele está suscetível a erro e quebrar a minha aplicação sem querer, eu posso quebrar fazendo uma validação manual.

[08:53] Porque eu posso inserir, eu posso criar uma nova classe de inserção e esquecer de fazer essa validação, eu posso, muitas vezes, fazer essa validação errada. Então o ideal é que não façamos isso. Então, como eu faço? Graças ao JDBC, nós temos uma maneira de criar - na verdade não criar, validar essas informações de forma que o JDBC me provê a solução.

[09:19] O tempo todo nós vínhamos criando Statements. Uma vez que criávamos um Statement, nós passávamos a Query concatenando valores ou , por exemplo no nosso caso do Insert, nós passávamos antes já o produto e a descrição direto, dentro do Values, como uma string. Só que agora, eu vou mudar esse comportamento.

[09:45] Invés de eu criar o Statement, eu vou preparar um Statement. Quando eu preparo um Statement, eu estou falando que a responsabilidade de gerenciar esses atributos que eu passo para a minha cláusula SQL, não vai ser mais eu que vou fazer, agora vai ser o JDBC. Então eu apago todas essas linhas anteriores, que eu tinha feito para exemplificar, do `Statement sql` no `TestaInsercaoComParametro`.

[10:16] Aqui, em `VALUES`, eu não vou ter mais nada disso, eu só vou falar o seguinte: dentro dessa minha cláusula SQL, esse meu Insert, ele vai receber dois parâmetros. Esses parâmetros serão o nome e a descrição. Como que eu vou saber, como que o `.prepareStatement` vai saber que essa interrogação e essa interrogação, em `VALUES (?, ?)`, vão ser substituídas pelo nome e a descrição?

[10:41] Quando estamos usando o `.prepareStatement`, ele vai nos retornar um `PreparedStatement`, e aqui começa a beleza da coisa. Quando eu quero *setar* esse atributo `String nome`, nessa primeira interrogação do `VALUE`, eu só preciso falar para ele o seguinte: `PreparedStatement`, *seta* uma string, no primeiro atributo, que vai ser o valor da variável `nome` - `stm.setString(1, nome);`.

[11:15] Fica bem legível, você bate o olho e você já consegue entender. Eu faço a mesma coisa para o segundo parâmetro de `VALUE`, para a segunda interrogação, só que agora eu falo `stm.setString(2, descricao);`. A maravilha disso é que quando eu preparo o meu Statement, eu estou falando para ele substituir essas interrogações de `VALUE`, pelo `nome` e `descrição`, eu não preciso mais me preocupar em passar o `stm` para o `execute`.

[11:47] Eu só preciso executar, a Query, ela já está preparada. Eu só preciso agora tirar esse `Statement.RETURN_GENERATED_KEYS` de dentro do `execute` também. Ele vai ficar `stm.execute();`, sem nenhum parâmetro e eu dou uma vírgula no final de `connection.prepareStatement("INSERT INTO PRODUTO (nome, descricao) VALUES (?, ?)", Statement.RETURN_GENERATED_KEYS);`.

[12:06] E falo que eu vou querer também que as chaves geradas sejam retornadas após a inserção. O código `TestaInsercaoComParametro` todo refatorado, compilando e agora nós só precisamos executar. Agora sim eu tenho que ter o meu ID gerado, de número 41. Vimos que o ID foi criado.

[12:32] Se eu listar os produtos, em `TestaListagem`, nós vamos ver que foi criado o produto da forma que nós tínhamos deixado, então agora, sem a nossa validação manual, sem precisarmos fazer intervenção naquele nome com aspas simples, o próprio `PreparedStatement`, ele já se preocupou e já identificou que é uma string, a aspa simples ela é uma string, foi um erro do usuário, ele não quer quebrar a aplicação, então ele vai tratar como string.

[13:06] A mesma coisa na descrição, ele não tratou o delete como um delete, ele tratou o delete como uma string. Então aqui está a maravilha do `PreparedStatement`. Além de ele evitar os SQL Injections, ele deixa o código bem mais legível, então vemos agora, se você bater o olho em `VALUES`, eu tenho dois atributos e embaixo eu *seto* esses atributos. Então ficou muito mais bonito, fica muito menos vulnerável o nosso código.

[13:37] Além de mais bonito, menos vulnerável, então é muito vantajoso. Com isso, agora nós estamos aptos a fazer o Refactory de todas as classes que nós criamos até agora. Então, para isso, a `TestaListagem`, a `TestaRemocao`, invés de usarmos o Statement, agora nós podemos passar a utilizar o `PreparedStatement`. Então é isso, aluno. Espero que tenham gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Dando continuidade ao nosso curso, após conhecermos o `PreparedStatement`, agora podemos alterar todas as nossas classes para utilizá-lo. Então, voltando à nossa classe `TestaListagem`, eu vou alterar o `Statement stm = connection.createStatement();` para `Statement stm = connection.prepareStatement(sql);`.

[00:27] E com o exemplo que fizemos de inserção, no `stm.execute("SELECT ID, NOME, DESCRICAO FROM PRODUTO");`, retiro a minha cláusula `sql`, o `("SELECT ID, NOME, DESCRICAO FROM PRODUTO");` e passo para dentro dos parênteses do `connection.prepareStatement`, ele agora é o responsável por preparar esse Statement.

[00:43] Diferente do exemplo da inserção, no `TestaListagem` nós não temos um atributo, mas se um dia eu precisar fazer algum `where`, que vai receber alguma condição, eu já tenho esse `prepareStatement` preparado e eu só preciso adicionar depois o atributo. Então, em vez de eu retornar um `Statement`, eu vou retornar um `PreparedStatement stm = connection.createStatement();`.

[01:19] E eu tiro o `import java.sql.Statement`, que não é mais utilizado. Como nós já vimos, se eu executar o `TestaListagem`, continua do mesmo jeito, nós não tivemos nenhum problema nesse Refactory, nessa alteração. Na classe `TestaRemocao` podemos fazer a mesma coisa. Então em vez de eu ter o `Statement stm = connection.createStatement();`, eu vou ter o `Statement stm = connection.prepareStatement(sql);`.

[01:53] E mais uma vez aqui, só para ficar claro, eu tiro a nossa string SQL `("DELETE FROM PRODUTO WHERE ID > 2)` de `stm.execute`, e passo para dentro dos parênteses de `connection.prepareStatement`. Para ficar mais interessante o nosso

exemplo, eu vou retirar o `2`, que estava fixo e colocar `("DELETE FROM PRODUTO WHERE ID > ?)`.

[02:14] E vou passar esse valor do ponto de interrogação através do `setInt`. Então eu vou fazer um `stm.setInt(parameterIndex, x);`, e vou passar nesse primeiro index, eu passo o valor `(1, 2)`. E continuo executando o delete normalmente. Se eu executar o `TestaRemocao`, agora ele vai me mostrar a quantidade de linhas que foram modificadas, duas linhas.

[02:49] Se nós voltarmos no nosso `TestaListagem`, não vai constar mais aqueles produtos que foram inseridos, que tinham os erros, caracteres especiais, enfim, aqueles testes que nós fizemos no `TestaInsercaoComParametro`. Então é isso, pessoal. Mais um Refactory feito no nosso código, espero que vocês tenham gostado e até a próxima aula.

07

O que aprendemos?

Nesta aula, aprendemos que:

- Ao executar SQL como `Statement`, temos um risco de segurança, chamado de ***SQL Injection***
  - *SQL Injection* nada mais é do que passar um novo comando SQL como parâmetro
- Para evitar *SQL Injection*, devemos usar a interface `PreparedStatement`
  - Diferentemente do `Statement`, o `PreparedStatement` trata (*sanitiza*) cada parâmetro do comando SQL

Transcrição

[00:00] Fala, aluno. Tudo bom? Anteriormente, nós vimos como utilizar o `PreparedStatement` no nosso código. Nós vimos que com ele é possível evitar os SQL Injections, que são capazes de acabar com a nossa aplicação. Vimos também que o nosso código fica mais sucinto, porque não precisamos mais ficar controlando a concatenação de variáveis com a nossa cláusula SQL.

[00:31] Enfim, todo aquele problema que tínhamos com validação de Query, enfim, temos várias vantagens em utilizá-lo. Porém agora eu quero continuar utilizando essa classe `TestaInsercaoComParametro`, porque agora, ao invés de adicionar um único produto, o produto que estará guardado nas nossas variáveis aqui, eu quero adicionar mais de um produto, eu quero ter opção de adicionar dois, três, quatro produtos, que for.

[01:00] Hoje o nosso código não está preparado para isso. Nós temos aqui dois `setString`, na verdade, que vão adicionar o nome e a descrição, que são os dois atributos que a nossa query espera. Vamos ter um método `execute();`, que executa essa cláusula SQL do insert e temos o método `getGeneratedKeys`, que retorna a chave daquele objeto que foi criado, aquele produto que foi criado.

[01:25] Então, para isso ficar melhor para a nossa refatoração, para adicionar mais do que um produto, eu vou extrair esse pedaço de código para um método. Se eu der um "Ctrl + 3" no Eclipse, no meu programa já está o "Extract Method", mas vocês podem procurar essa mesma opção, que vai ter no programa de vocês.

[01:47] Eu quero colocar o nome do método de "adicionarVarivel". Ele já está chamando o método e passando os atributos que nós vamos precisar, que é o `nome`, para *setar* no `stm.setString(1, nome); ; descricao` para *setar* no segundo

`setString`, o `stm.setString(2, descricao)`, e o Statement, o `stm`, que é o que vai de fato executar a nossa cláusula SQL, que vai retornar as chaves geradas, enfim, nós precisamos desses três atributos.

[02:19] Para ficar melhor o nosso código, eu não vou passar o `String nome = "";` e o `String descricao = "";` como valor na variável, eu vou passar diretamente as strings na `adicionarVariavel("", "", stm);`. Vou copiar esse trecho da `adicionarVariavel`, porque agora eu quero adicionar dois produtos.

[02:37] Então, o primeiro produto eu quero uma `("SmartTV", "", stm);`, eu vou botar `("SmartTV", "45 polegadas", stm);`. E vou botar um `("Radio", "Radio de bateria", stm);`. Vocês podem dar o nome que vocês quiserem para os produtos, é só para exemplificar mesmo. Com esse nosso código, não é para ter nenhuma diferença dos outros códigos que nós já viemos executando apenas com um produto.

[03:12] Ele tem que adicionar os dois produtos e nos retornar duas chaves. Vamos verificar? Vou mandar executar a minha aplicação. Beleza, olha, ele criou dois IDs para mim, o 68 e o 69. Para conferirmos se os produtos foram de fato inseridos, vamos na `TestaListagem` e vamos verificar se os dois produtos estão lá. Já garantimos que nós temos os dois produtos novos inseridos na base.

[03:21] Só que nós mandamos executar aqui a classe `TestaInsercaoComParamentro` e ele adicionou os dois produtos. O que aconteceria se no momento em que eu estivesse adicionando o produto rádio, eu tivesse uma exceção? Qual seria o comportamento da minha aplicação? Ela adicionaria apenas o primeiro produto ou ela não adicionaria ninguém, por conta da exceção que deu na hora de adicionar o segundo produto?

[04:13] Qual seria esse comportamento? Para descobrirmos, eu vou forçar um pouco no código, para ele de fato dar uma exceção. Então eu vou verificar se o nome do produto é igual à `Radio`, com `if(nome.equals("Radio"))`, porque se for, eu vou dar uma `throw new RuntimeException("");`, eu vou forçar aqui um erro. E vou botar uma mensagem que `("Não foi possível adicionar o produto");`.

[04:47] Eu vou mandar agora remover esses dois produtos que nós inserimos, com `stm.setInt(1, 2);` em `TestaRemocao`. Apagamos os produtos. Agora, em `TestaInsercaoComParametro` eu vou mandar novamente inserir os meus dois produtos. Quando eu mandar executar a classe, o ID criado foi o 70, mas estourou uma exceção, informando que não foi possível adicionar o produto.

[05:17] Agora para eu descobrir qual foi o comportamento do meu sistema, se ele adicionou só o primeiro, se ele não adicionou ninguém, vamos listar novamente no `TestaListagem`. Vou dar um "Run As > Java Application". Ele adicionou o primeiro produto, como ele tinha informado no momento da inserção, só que o segundo o produto não consta aqui.

[05:37] Então o que a nossa aplicação está fazendo? Então ela está pegando o primeiro produto que pedi para inserir, ela abre uma transação, vai concatenar o nome e a descrição na minha cláusula SQL, vai mandar executar essa cláusula, vai dar o `ResultSet` e fecha a transação. No segundo produto, ela vai abrir uma nova transação, vai concatenar o nome e a descrição do produto.

[06:18] Só que vai dar uma exceção, porque o nome é `Radio` e falamos que se fosse `Radio` tinha que dar uma SQL Exception. Mas olha só: nós abrimos uma transação, fechamos uma transação. Abre uma transação, fecha uma transação. A meu ver, isso poderia estar de uma forma melhor no nosso código. Então, o que acontece?

[06:43] Eu posso assumir o controle dessa transação. Eu não quero mais que o JDBC faça dessa maneira. Porque quando eu tenho o controle da minha transação, no momento em que der erro, eu posso não incluir o segundo produto que deu erro, mas também posso falar para ele apagar o primeiro produto que foi inserido, digamos que uma transação única, enfim. Para fazer isso, o que eu preciso fazer?

[07:19] Eu tenho que tirar a responsabilidade, o Commit, ou seja, da inserção do meu produto, das mãos do JDBC. E eu vou fazer o seguinte: `connection.setAutoCommit(false);`, vou falar: `connection`, o `setAutoCommit` agora vai ser *false*, ou seja, eu vou controlar o momento do Commit da minha aplicação, no momento da minha transação. Então com esse `connection.setAutoCommit(false)`, agora eu que controlo a minha transação.

[07:48] Eu posso então, agora, fazer o seguinte: eu vou apagar o meu produto "SmartTV" que foi adicionado, em `TestaRemocao`, e eu vou adicionar de novo, em `TestaInsercaoComParametros`. Vamos ver o comportamento agora com esse `connection.setAutoCommit(false);`. Ele criou o ID 71, deu erro, um cenário bem parecido com o anterior.

[08:13] Vamos ver o que aconteceu, aqui na nossa `TestaListagem`. Não incluiu nenhum produto. Então vamos fazer o seguinte, eu vou comentar aqui essa exceção, o `if(nome.equals("Radio")`, que ele pode dar por conta dessa exceção. Então vou comentar aqui. Agora vou novamente mandar inserir e vamos ver o resultado.

[08:47] O ID criado foi o 72, o ID criado foi o 73. Um cenário igual ao que estava no começo da aula. Deixa eu testar no `TestaListagem`. Não adicionou nenhum dos dois. Bom, temos o seguinte cenário: agora nós tiramos a responsabilidade do JDBC de fazer os Commits das nossas transações, de fato inserir o produto na nossa tabela. Só que agora, com isso, não estamos conseguindo mais adicionar nenhum produto.

[09:28] Por que será que isso está acontecendo? Resolvemos um problema, porém agora nós temos que tratar esse segundo problema. Isso nós já vamos ter insumo para na próxima aula já tratar esse problema e verificar o motivo de não estar inserindo esse produto e como nós vamos tratar isso. Então espero que vocês tenham gostado dessa aula. Até o próximo vídeo.

Transcrição

[00:00] Fala, aluno. Tudo bom? Voltando ao nosso curso de JDBC, agora eu quero voltar na nossa classe `TestaInsercaoComParametro`, onde nós vimos que estava tendo um erro na hora de inserir os produtos. Nós vimos que foi *setado* o `setAutoCommit(false);`, que é exatamente para nós termos o controle da nossa transação, agora eu que quero controlar a transação do início ao fim na hora de adicionar ao banco de dados.

[00:30] Porém não estava sendo inserido. Qual é a ideia da nossa transação? Para mim, ela só vai valer se eu adicionar esses dois produtos ou nenhum deles. Por que eu bato nessa tecla, de adicionar os dois ou nenhum? Vamos pensar em uma conta, uma conta bancária mesmo, onde eu tenho uma conta de origem e tenho uma conta de destino. Vamos supor que na conta de origem nós vamos transferir um dinheiro para a conta de destino, e essa conta destino vai receber.

[01:06] Então, ou seja, eu transfiro o dinheiro, o dinheiro sai da minha conta de origem e chega na conta de destino, então a conta de destino recebe. Se no momento em que eu estiver fazendo essa operação, quando estiver chegando na conta de destino, vamos supor que foi encontrado um erro e a transação, a nossa operação, ela não foi concluída. Ou seja, o dinheiro não chegou na conta de destino.

[01:37] Essa transação, na verdade, ela vai ter que ser desfeita, o dinheiro vai ter que voltar para a conta de origem e vou ter que receber, no meu canal de autoatendimento, uma informação: seguinte, a sua transação não foi realizada com sucesso, por favor tente novamente mais tarde. Algo nesse nível, porque para mim não faz sentido uma transação sair do meu banco, sair da minha conta, sair um dinheiro da minha conta e não chegar na conta de destino.

[02:12] Mas, imagina, se tiver já saído da minha conta, uma transação para sair e uma outra transação para chegar. Então não fazia sentido, porque saiu da minha conta com sucesso, mas não chegou na outra. O dinheiro ficou onde? Então por isso que eu fico batendo nessa tecla de que as nossas transações, elas têm que ser daquela maneira.

[02:34] Os dois produtos, eles têm que ser inseridos juntos, pelo mesmo motivo de uma conta bancária, que tem que ser na mesma transação, porque se eu tiver qualquer erro em uma das pontas, a transação, ela tem que ser desfeita. Os bancos de dados, hoje em dia, eles já estão trabalhando, eles já estão preparados para essas transações. Eu tenho um encadeamento de comandos SQL, de operações SQL, que se alguma delas der errado, todas as outras desfazem.

[03:00] Eu tenho serviços, hoje em dia, em sistemas, que ou você faz todo aquele procedimento ou não faz nenhum, volta todo mundo. Então por isso que eu venho batendo nessa tecla de nós termos esses dois produtos sendo inseridos juntos, é esse o conceito de transação. Entendemos o conceito, como que agora funciona para resolvermos o nosso problema?

[03:27] Quando estamos trabalhando com controle transacional manual, que é quando botamos o `.setAutoCommit(false);`, o Commit nós temos que explicitar no nosso código. Então em `TestaInsercaoComParametro`, dentro de `Connection`, eu vou ter um método chamado `commit();`.

[03:46] Agora esse `connection.commit();` que vai fazer o seguinte: quando todos os Statements `adicionarVariavel` estiverem preparados, ou seja, quando eu já tiver falado para a minha transação que nós vamos ter um produto que vai ter o seu método `execute`, que vai me retornar as `getGeneratedKeys`, outro produto que vai chamar o `execute`, que vai me retornar as `getGeneratedKeys`, quando tudo estiver ok e não tiver me dado problema, então dá o Commit na minha transação.

[04:15] Agora, como nós já comentamos aqui o `if(nome.equals("Radio"))`, se eu executar o `TestaVariavelComParametro`, ele tem que dar certo. Vamos verificar o resultado. Foram criados dois IDs, o 103 e o 104. Para testarmos se os produtos foram de fato criados, vamos mandar executar a `TestaListagem`. Estamos aqui, nós temos agora o produto com IDs 103, que é uma "SmartTV" e o produto com IDs 104, que é um "Radio de Bateria".

[04:47] Então agora a nossa classe `TestaInsercaoComParametro` está funcionando perfeitamente. Só que veja bem, eu tenho como melhorar esse meu código. Eu posso, na verdade, quando eu realizar o Commit na minha transação, se tiver algum erro, eu posso dar um rollback, ou seja, eu posso explicitar o rollback falando o seguinte: desfaz todo mundo e me dá uma mensagem.

[05:11] Para termos esse controle mais fino da transação, eu vou abrir um `try`, em `TestaInsercaoComParametro`, e vou esse código do `PreparedStatement`, `adicionarVariavel`, `connection.commit`, `stm.close` e `connection.close`, que nós tínhamos do lado de fora para dentro do `try {}`.

[05:25] Agora eu vou pegar uma `catch (Exception)` e vou falar o seguinte: me fala qual foi a exceção. Eu quero que dê uma mensagem, falando que o `System.out.println("ROLLBACK EXECUTADO");` e eu quero fazer de fato o `connection.rollback();`, que é isso que consiste o nosso `.setAutoCommit(false);`, é nós controlarmos a nossa transação, em que momento ela vai dar o rollback, em que momento ela vai dar o Commit.

[05:59] Então é esse tipo de estratégia que vai servir para termos esse controle da nossa transação, quando *setamos* o `setAutoCommit(false);`. Para verificarmos como ficou o nosso código, eu vou tirar o comentário do nosso `if` porque agora eu quero que dê um erro no momento em que estivermos inserindo, preparando os nossos Statements.

[06:26] Só para ficar mais claro, eu vou apagar esses dois produtos que acabamos de inserir. Duas linhas foram modificadas, duas linhas foram apagadas. Para termos uma confirmação, vamos executar `TestaListagem` de novo, o nosso código. Agora só os dois produtos iniciais. Agora vamos executar o nosso código `TestaInsercaoComParametro`, com o controle mais fino da nossa transação.

[06:56] Executou. Olha lá, ele fala que o ID foi criado, o 105, mas nós já tínhamos visto isso anteriormente. Ele falou qual foi a motivação, qual foi o *print stack trace* da minha sessão e ainda me falou "ROLLBACK EXECUTADO". Com isso eu estou garantido agora que ele vai entrar nesse momento do `catch (Exception)`, e vai fazer o `connection.rollback();`, porque ele executou o `e.printStackTrace();` e executou a nossa mensagem.

[07:35] Então com isso, agora estou garantindo que toda a minha transação, ela está daquela forma que nós queríamos: ou vai adicionar todo mundo ou não vai adicionar ninguém, porque nós vimos, o primeiro produto, ele não passou no `if`, ele não geraria uma exceção. Anteriormente, como estava com o `setAutoCommit` igual a true, ele estava Commitando a cada transação, agora não.

[08:02] Agora eu tenho a minha transação realizando o Commit só depois que der tudo certo, ou então ele vai sair abruptamente no `if`, vai cair no `catch`, que vai capturar essa exceção, vai me mostrar qual foi a motivação da exceção, me mostra a mensagem do `println` e de fato deu o rollback. Agora não tenho nenhum problema com a minha transação.

[08:23] Então, com isso, nós estamos vendo como funciona esse controle transacional já na vida real. Esses aqui vão ser códigos, no dia a dia, no trabalho, que serão feitos dessa forma. Então agora nós já temos um código com um controle mais fino, um código mais complexo, mas mais bonito também.

Transcrição

[00:00] Fala, aluno. Tudo bom? Anteriormente nós vimos como fazer o controle das nossas transações de forma mais fina, de forma mais bonita, por assim dizer. Nós vimos que uma transação, ela vai envolver uma ou mais operações, então no nosso caso, é a adição de dois produtos na mesma transação.

[00:24] E com aquela velha história, ou ele vai adicionar os dois produtos ou ele não vai adicionar ninguém, porque se ele der alguma exceção, que é caso do nosso exemplo do `if`, ele vai desfazer a operação da primeira inserção, e ficamos com uma transação bem certa, conforme o que esperamos de uma transação mesmo.

[00:55] Só que uma coisa que nós vamos começar a reparar no código, são por exemplo esses Statements, que eles devem ser fechados. Nós temos uma conexão que quando abrimos, deve ser fechada. Temos um `PreparedStatement` que quando abre, ele deve ser fechado. Tem o `ResultSet` também na mesma situação. E hoje nós vemos que é fácil de esquecer de explicitar o close desses Statements.

[01:29] Esse é um problema, porque você esquece, então ele depende de ser explícito, esse fechamento. Então, no caso de uma conexão, ele pode abrir várias conexões e eu esquecer de fechar, isso pode trazer problemas de performance no futuro, pode trazer problemas de banco de dados no futuro. Esse é um problema.

[01:52] Só que nós temos outro problema também. Por exemplo, no nosso `catch`, que recuperamos a exceção caso aconteça alguma coisa na transação, ele vai fazer o `rollback` dentro do `catch`. Mas, por exemplo, dependendo de como eu tratar a minha exceção, ele pode nunca sair desse `catch`, pode nunca fechar o close.

[02:10] Então eu posso até ter nunca esquecido de fechar a minha `connection`, só que o fato de eu tratar errado a minha exceção dentro do meu `catch`, ele pode nunca sair do `catch` e eu posso ter o mesmo problema que eu teria se eu tivesse esquecido de fechar a minha conexão. Então, para isso, eu tenho, a partir do Java 7, um recurso que chama "try-with-resources", que é o try com recursos.

[02:39] Esse try com recursos, ele serve para nos auxiliar nos fechamentos desses Statements que precisamos explicitar que sejam fechados. Se verificarmos no `PreparedStatement`, ele vai estender o Statement e também que o Statement vai estender um `AutoCloseable`. Esse `AutoCloseable` é o que tem o nosso método `close()`.

[03:07] Com o "try-with-resources", quando eu abro o `try`, eu por exemplo, posso colocar o `PreparedStatement stm = connection.prepareStatement("INSERT INTO PRODUTO (nome, descricao) VALUES (?, ?)", Statement.RETURN_GENERATED_KEYS);` entre os parênteses do `try()`.

[03:26] Com essa ação que eu acabei de fazer, e pelo o `PreparedStatement`, ele estender de `AutoClosable`, eu não preciso explicitar o fechamento desse `PreparedStatement`, porque ao final do `try`, quando eu terminar a execução do meu bloco `try`, ele está garantido que o Statement será fechado.

[03:51] Então, com essa ação de colocar o bloco do `PreparedStatement` entre os parênteses do `try()`, eu não preciso mais me preocupar agora em ficar explicitando os meus fechamentos. Então a mesma situação nós podemos fazer no `ResultSet`. Invés de eu fechar o `ResultSet` explícito em `rst.close();`, no final do bloco, eu vou abrir antes de `ResultSet` um try com recursos.

[04:21] E eu só preciso substituir o `rst.close();` por uma chave, por um fechamento da minha chave, do meu bloco do `try`. Aqui eu já estou garantindo que o recurso do `ResultSet` será fechado pois ele estende também o recurso `AutoCloseable` do `PreparedStatement`. Podemos também fazer a mesma coisa com a nossa `Connection connection = factory.recuperarConexao();`.

[04:41] Então vou abrir `try(Connection connection = factory.recuperarConexao());{` , os parênteses do meu `try` , vou botar o recurso dentro do `try` , que é o recurso da `Connection` , da minha conexão e no `connection.close();` , no seu fechamento, invés de eu explicitar o `.close` , eu só vou apagar a linha e fechar as chaves. Deixa eu abrir a chave no `try` do `Connection` .

[05:06] Pronto, com isso, agora eu tenho um código mais enxuto, porque nós temos menos linhas de código, e nós não temos como - quer dizer, não é que nós não temos, é que fica mais fácil, porque não precisamos agora explicitar esses fechamentos e não ficamos com aqueles problemas de um esquecimento de fechar uma conexão.

[05:34] Ou então até menos na hora de tratar uma exceção ele não sair do bloco, enfim, aqueles problemas que nós estávamos falando no começo da aula. Então o "try-with-resources" vai servir para isso. Além de ficar com um código mais bonito, ele vem com esse benefício de tirar da nossa responsabilidade explicitar esses fechamentos. Então é isso, pessoal. Espero que tenham gostado e até a próxima aula.

09

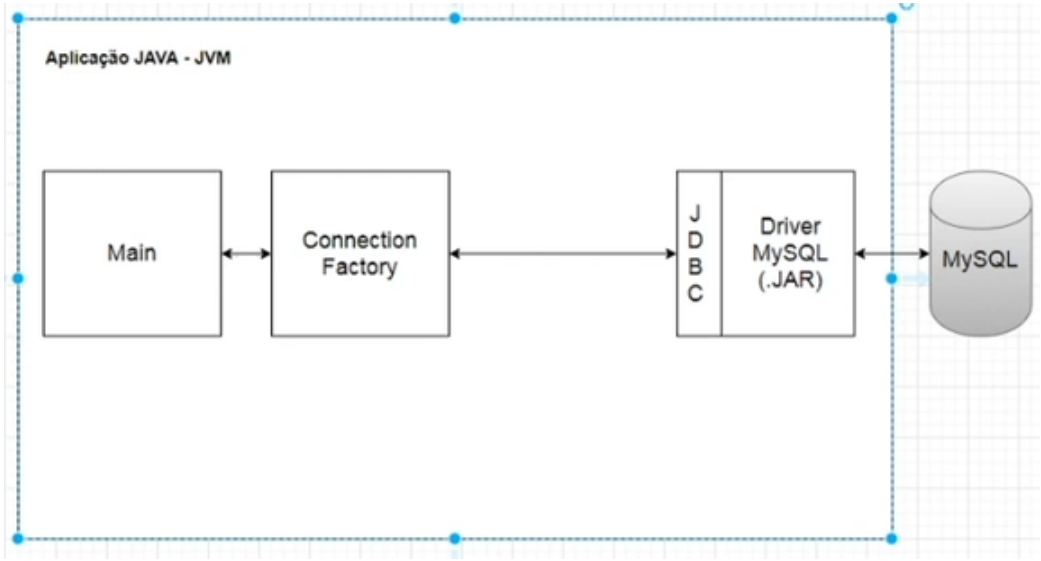
O que aprendemos?

Nesta aula, aprendemos que:

- O banco de dados oferece um recurso chamado de **transação**, para juntar várias alterações como unidade de trabalho
  - Se uma alteração falha, nenhuma alteração é aplicada (é feito um *rollback* da transação)
  - Todas as alterações precisam funcionar para serem aceitas (é feito um `commit`)
- `commit` e `rollback` são operações clássicas de transações
- Para garantir o fechamento dos recursos, existe no Java uma cláusula *try-with-resources*
  - O recurso em questão deve usar a interface `Autoclosable`

Transcrição

[00:00] Fala, aluno. Tudo bom? Dando continuidade ao nosso curso de JDBC, eu gostaria de revisar agora como está a estrutura da nossa aplicação. Nós estamos comunicando a nossa aplicação em Java com o banco de dados MySQL. Só que nós vimos que essa aplicação, ela não se comunica com o banco de dados de forma nativa.



[00:20] Justamente porque de um lado nós temos a linguagem Java e do outro lado, nós temos uma linguagem MySQL, que não nos importa qual é, mas nós sabemos que não é simplesmente codificar nas nossas classes que nós vamos conseguir comunicar com o MySQL. Então, para isso, nós vimos que é necessário um driver.

[00:46] Esse driver é específico dos bancos de dados. O que isso quer dizer? Hoje, utilizando o MySQL como base dados, nós vamos ter que utilizar um driver do próprio MySQL. Se um dia eu quiser trocar o meu banco de dados de MySQL para

Postgre, eu vou ter que utilizar aqui um driver do Postgre.

[01:04] Só que isso já nos levantou uma outra dúvida: se eu tenho um driver específico, então quando eu for trocar o MySQL por um Postgre, então eu vou ter que sair procurando no código onde é utilizada essa implementação do driver, onde estamos chamando as classes, interfaces do driver, e vamos ter que alterar para classes e interfaces do Postgre.

[01:31] Só que nós vimos que não é necessário, porque nós temos uma interface, que é chamada JDBC, que ela tem as suas classes e interfaces, que expõe essa implementação dos drivers. Então quando eu quero solicitar uma conexão para o MySQL, eu vou na minha interface JDBC e eu solicito um `getConnection`, que é uma interface do próprio JDBC e ele que se vire com o driver e faça a requisição da minha conexão.

[02:08] Então, com a interface do JDBC, nós vimos que não precisamos conversar diretamente com as classes dos drivers, que são bem específicas. Com isso, nós vimos então que quando eu quero chamar, quando eu quero abrir uma conexão com o banco de dados o meu Driver manager, o meu get connection e passar a string de conexão entre parênteses. Só que nós vimos que quando necessitávamos abrir uma conexão, nós tínhamos que repetir esse código.

[02:48] Voltando à parte de se um dia precisássemos alterar o nosso banco de dados, em todo lugar que fosse chamado o driver manager e o `.getConnection`, eu teria que sair alterando a string de conexão, porque eu agora me conecto com outro banco. E nós vimos que isso não é muito legal, porque poderíamos esquecer, enfim, tudo aquilo que já vimos que poderia prejudicar a nossa aplicação.

[:03:14] Então por isso construímos uma Connection Factory. Essa Connection Factory é justamente para isolar a chamada do Driver manager em um único lugar e expor um método recuperar conexão para os nossos main, que de fato vão ser os `TestaInsercao`, `TestaListagem`, tudo aquilo que viemos fazendo ao longo do curso.

[03:36] Revisada estrutura da nossa aplicação, agora temos que nos atentar a um problema. Não a um problema, mas a uma situação. Hoje, toda requisição que estamos fazendo está sendo apenas de uma única conexão. Então quando eu chamo o

meu main, ele vai, abre uma conexão com o banco de dados e me retorna o que eu quero, fechei a conexão. Quando eu faço outro main, ele faz o mesmo procedimento e fecha a conexão.

[04:06] Só que vamos agora no site da Alura. Se eu fizer o *login* e entrar no meu Dashboard, vamos supor que eu quero ver quais são as formações. Quando eu entro, por exemplo, nas formações em mobile de Android, quando eu estou fazendo essas requisições de página, a plataforma faz a solicitação ao banco de dados para me retornar essas informações da página para eu conseguir ver tudo isso que estamos vendo na nossa tela.

[04:39] Só que no momento em que eu estou na página, vamos supor que eu tenho um outro aluno também esteja fazendo a mesma coisa. Voltando ao nosso diagrama, como seria o funcionamento desse caso na nossa aplicação? Eu fiz uma requisição e o meu banco de dados está processando. O outro aluno, ele fez uma requisição, então eu vou enfileirar a requisição dele para só quando a minha for processada a dele seja processada?

[05:06] Faz muito sentido? Para mim não faz. Então o que poderia fazer mais sentido? No momento em que eu fiz uma conexão, outro aluno fez também, então nós abrimos duas conexões. Dessa maneira já fica um pouco melhor, porque agora eu não estou enfileirando as conexões e o meu banco de dados está processando as duas requisições ao mesmo tempo.

[05:34] Só que temos agora, no site da Alura, uma parte de novidades. Vamos supor que disparou um e-mail falando sobre um curso de Build de uma aplicação .NET. Esse curso teve um alcance muito grande, vários alunos gostaram do que se propõe a fazer no curso, e vários alunos começaram a fazer requisições na plataforma para conhecer mais sobre o curso. Eles receberam o e-mail mas foram na plataforma da Alura para conhecer mais sobre o curso.

[06:09] Então a tendência é que essas conexões entre a aplicação e o JDBC, elas vão aumentando. O alcance de uma plataforma como a Alura é de milhões de alunos. Quando eu tenho milhões de alunos, com essa estrutura de abrir conexões descontroladamente, eu vou abrir milhares de conexões. E o que vai acontecer com o meu banco de dados?

[06:33] Ele não vai aguentar, porque, de fato, o MySQL é um banco de dados muito bom, é um dos mais utilizados hoje em dia, nós já vimos sobre isso. Só que nenhum banco de dados vai aguentar um processamento de milhões de conexões de

uma vez, de forma descontrolada. Aqui nós encontramos um problema. Como solucionamos esse problema?

[06:58] O ideal seria então que invés de eu sair abrindo conexões de forma descontrolada, antes da minha interface JDBC, eu tivesse uma outra caixa no diagrama que fizesse o seguinte: quando eu subir a minha aplicação, eu quero ter um número X de conexões abertas. Eu não quero que seja apenas uma, porque isso nós vimos que pode enfileirar as nossas requisições.

[07:37] Mas eu também não quero que ele abra de forma descontrolada, porque isso pode fazer com que o nosso banco de dados, ele caia. Então eu quero agora ter um número X de conexões estabelecidas. Com essa caixa entre a aplicação e o JDBC, o que ela faria? Ela seria responsável por abrir essas conexões com o banco de dados.

[08:06] Então agora invés de ter todo aquele descontrole de conexões, agora eu tenho um número X, que eu posso até colocar um número mínimo e um número máximo de conexões e essa caixa que vai conversar agora com a interface JDBC. Essa estrutura agora parece ficar muito legal, porque passamos a ter o controle dessas conexões, a nossa aplicação fica de uma forma bem legal.

[08:42] Agora, o que precisaríamos fazer? Nós precisaríamos, de fato, implementar essa caixa, que até então não sabemos como seria o código. Só que olha que maravilha: nós não vamos precisar implementar essa caixa, porque ela já existe hoje para nós, ela é chamada de Pool de conexões.

[09:10] Então agora, essa caixa entre a aplicação e o JDBC, nós vamos ter um Pool de conexões, que é exatamente para isso que estamos vendo, é para ter um número de conexões abertas para nós. Deixa eu só aumentar a fonte para ficar mais fácil de ver, para ficar bonito o nosso desenho. Então agora com esse Pool de conexões, deixa eu só abaixar ele, para não ficar em cima.

[09:41] Então é essa caixa aqui. O Pool de conexões, para nós, vai ser também um driver, assim como nós temos o driver do MySQL, nós temos o driver do Pool de conexões. O driver que nós vamos utilizar, para o nosso Pool de conexões, para se comunicar com o MySQL, vai se chamar C3P0.

[10:05] Ele não é o único, eu tenho vários drivers, assim como nós temos os drivers de banco de dados, o MySQL, o Postgre, eu vou ter também vários drivers de Pool de conexões. Só que o C3P0, ele vai nos servir para tudo o que vamos precisar usar na hora que formos implementar o nosso código de fato.

[10:34] A documentação dele não é complexa, a documentação dele é tranquila, nós encontramos na internet tudo o que precisa para configurar, para termos um Pool de conexões correto, de acordo com as boas práticas, enfim. Só que se o Pool de conexões, ele é um driver e ele pode ser alterado, eu posso mudar o C3P0 para outro driver, vamos ter aquele problema que teríamos sem a interface JDBC.

[11:26] Porque eu vou implementar um Pool de conexões, eu posso injetar ele em alguns pontos na minha aplicação, mas quando eu fosse alterar ele, eu teria que sair modificando, talvez, na minha aplicação onde eu chamo esse meu Pool de conexões. Só que temos uma vantagem: como temos a interface JDBC, nós temos também a interface para o nosso Pool de conexões, que se chama Datasource.

[11:58] Então deixa eu só criar um retângulo para o diagrama, que ele vai ficar um pouco antes do Pool de conexões. Então esse retângulo, ele vai ser a minha interface do Pool de conexões. Essa interface vai ser a minha interface Datasource. Deixa eu escrever, Datasource. Com o Datasource, vamos conseguir expor todas as configurações do nosso Pool de conexões.

[12:41] Como eu tinha falado anteriormente, no Pool de conexões eu posso limitar, pôr uma quantidade mínima de conexões abertas e uma quantidade máxima. Então eu faço todas essas configurações no meu Pool de conexões e com o meu Datasource, eu apenas exponho apenas para a minha Connection Factory.

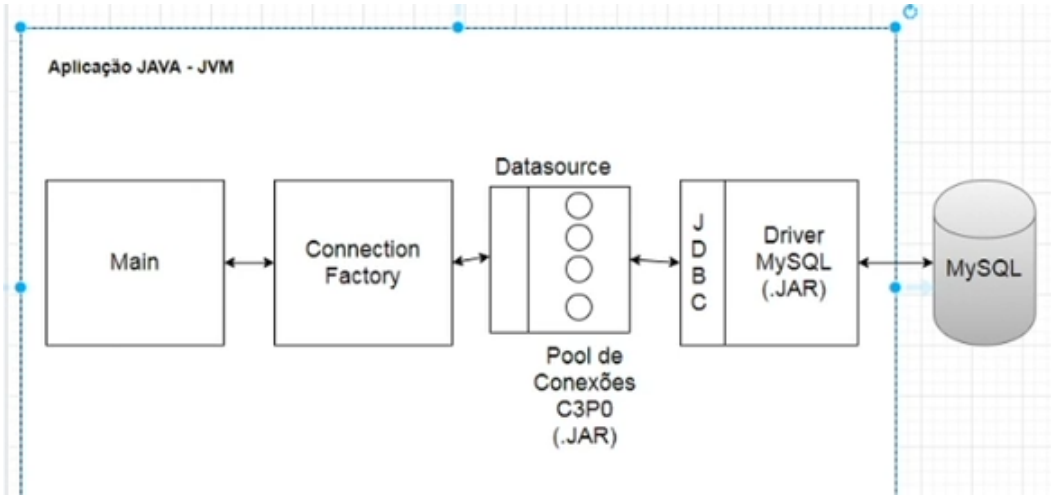
[12:58] Com essa estrutura agora, eu não ia mais precisar solicitar da Connection factory a abertura de uma conexão, eu só ia ter que perguntar para o meu Datasource o seguinte: Datasource, eu tenho conexão aberta? Tenho. Então eu vou utilizá-la. E a nossa Connection factory passa a fazer esse tipo de trabalho.

[13:28] Então qual é o objetivo do Pool de conexões? Eu ter um controle maior das minhas conexões, de quantas conexões vão estar abertas, porque nós vimos que não é uma boa prática eu abrir e fechar conexões descontroladamente e eu também

não vou ter apenas uma conexão, para enfileirar as minhas requisições. Então, com essa estrutura de Pool de conexões, já começamos a ver como funcionam aplicações do mundo real.

[13:59] Então todas as aplicações que vão trabalhar com o banco de dados hoje, na verdade usam um Pool de conexões com interface Datasource. Então agora a intenção é que nas próximas aulas, nós vamos botar a mão na massa mesmo, que vai ser para exatamente implementarmos essa caixa do Pool de conexões e o Datasource no nosso código. E vamos verificar como fazer as configurações do Pool de conexões e vamos ver como é a interface Datasource.

[14:33] Então, nessa aula nós vamos ficando por aqui. Espero que vocês tenham entendido o desenho. Quem ainda está com alguma dúvida, na hora que formos programar, que formos codificar, eu acho que vai sanar essas dúvidas. A ideia agora é que a nossa aplicação agora, ela já vá para um patamar de uma aplicação real. Então é isso, aluno. Vejo vocês no próximo vídeo.



Transcrição

[00:00] Fala, aluno. Tudo bom? Agora já com o conhecimento adquirido de como funciona um Pool de conexões, nós vamos partir de fato para a implementação na nossa aplicação. Hoje, o que nós fazemos é retornar uma conexão seca, então quando é feita a requisição que precisa recuperar uma conexão, ela vai recuperar a conexão com o nosso banco de dados, vai fechar e quando vier outra requisição, ela vai abrir uma nova requisição.

[00:27] Nós vimos que isso, com o tempo, com aplicações de grande complexidade, com processamento de muitos dados, isso pode ficar complicado no futuro, porque pode onerar o nosso banco de dados. Então para conseguirmos implementar o nosso Pool de conexões, nós vamos ter que inserir dois JARs externos à nossa aplicação clicando em "loja-virtual-repository" com o botão direito e selecionando a opção "Properties" para abrir a janela "Properties for loja-virtual-repository".

[00:52] Selecionando "Classpath" e clicando no botão "Add External JARs...", esses arquivos serão o "mchange-commons-java-0.2.16" e o "c3p0-0.9.5.4". Esses dois JARs, eles estarão disponíveis no repositório da Alura para vocês baixarem. Eu peço encarecidamente que vocês usem esses dois JARs, que estão no repositório, porque são os que estamos usando aqui na aula.

[01:13] Podemos utilizar um mais novo? Pode, só que pode ter problema de compatibilidade, o nosso problema pode ser por conta de versão, e essa não é a intenção. A intenção aqui é praticarmos esses conhecimentos que estamos adquirindo do JDBC. Então vamos evitar utilizar outra versão, vamos usar as que estão no repositório, que vai dar tudo certo.

[01:38] Vou aplicar aqui os arquivos. Então, como foi falado, na aula em que nós fizemos o desenho de como funciona um Pool de conexões, a ideia é que agora eu não fique abrindo e fechando conexões com o meu banco de dados. Mas também eu tenho que ter um meio termo, também não posso enfileirar as minhas conexões e esperar que eu só tenha uma conexão.

[02:01] Então vamos ter que criar esse Pool de conexões mesmo. E qual é a ideia? Toda vez agora, no meu `public ConnectionFactory`, toda vez que ele for instanciado, eu quero que o meu Pool de conexões, ele seja instanciado também, que ele me dê um Pool de conexões, na verdade.

[02:19] Então, com a C3P0, eu tenho um `comboPooledDataSource`. Esse `comboPooledDataSource`, nós vamos instanciar ele também para podermos fazer algumas configurações nele. Então deixa eu fazer: `ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();`.

[02:48] Antes de continuarmos, para não ficar tão obscuro para nós o porquê utilizar esse JAR, essa biblioteca, nós vamos no site da biblioteca C3P0, que é esse "mchange.com/project/c3p0/". Ela vai explicar que o C3P0, ele é uma *library*, é um JAR que faz o que os drivers JDBC, hoje eles fazem.

[03:19] Então, por exemplo, eu falei para vocês que o SQL Server vai ter uma implementação para Datasource, o Postgre vai ter um driver que vai ter uma implementação para o Datasource. No caso do MySQL, precisamos utilizar esse C3P0 para implementar o nosso Pool de conexões para ser exposto pela nossa interface Datasource, então esse é o principal objetivo do C3P0, por isso estamos utilizando ele.

[03:52] Então vamos lá, continuando a configuração do nosso Pool, eu tenho que *setar* a `(jdbcUrl)`, que nada mais é do que a nossa string de conexão que já tínhamos usado anteriormente, quando estávamos retornando a conexão com o Driver manager. Então vamos só copiar a URL e colar dentro dos parênteses de `comboPooledDataSource.setJdbcUrl();`.

[04:19] Nós temos também que *setar* o usuário com `comboPooledDataSource.setUser("");`, vai ser o `("root")`. E vamos *setar* o nosso password, com `comboPooledDataSource.setPassword("");`, que é `("root")` também. Quando nós estávamos na aula do desenho, nós falamos que nós configuramos o Pool de conexões e quem expõe ela, temos que implementar uma interface Datasource para poder expor essas informações para a nossa aplicação.

[05:00] Então o que nós vamos fazer aqui é criar uma variável que vai ser do tipo Datasource. Esse Datasource é como foi falado anteriormente, ele não é de um driver específico, igual é o `ComboPooledDataSource`, ela é do Java, então o Datasource é

uma interface que será implementada por esses drivers, que têm por objetivo expor mesmo essas informações do Pool.

[05:32] Então, quando eu crio esse `public DataSource dataSource;`, eu posso fazer o seguinte, eu posso falar `this.dataSource =` recebe o `= ComboPooledDataSource;`, que eu estou falando o seguinte: eu configurei aqui o meu Pool de conexões e agora, Datasource, expõe isso para a minha aplicação. Então é a nossa Datasource que vai fazer com que o nosso Pool, ele funcione, ele seja criado de acordo com aquilo que vimos no nosso desenho.

[SCREENSHOTS]

[06:09] Uma vez que eu tenho esse `ComboPooledDataSource` configurado, eu já não vou mais querer prover uma conexão com o `return DriverManager`, eu vou então tirar essa parte do código e o que eu vou fazer é o seguinte: vou chamar o meu `return this.dataSource`. Dentro desse `DataSource`, se nós entrarmos nele, nós vamos ver que ele tem um `getConnection()`.

[06:32] Então eu estou falando o seguinte: pronto `DataSource`, pega para mim a conexão que está disponível no Pool de conexões. Então com isso nós começamos a ter mais ou menos aquela ideia de como funciona o desenho. Então vai vir uma requisição, ele vai pedir uma conexão, e essa conexão, ela vai estar disponível naquele Pool de conexões, eu não vou precisar abrir uma conexão direta com o nosso banco de dados.

[07:02] Então essa é a principal diferença para a nossa aplicação agora. Então em vez de eu ir no banco de dados eu ir direto depois que acabar o processamento da minha requisição, eu mato essa conexão? Não, eu vou reaproveitar essa conexão, uma que já esteja aberta, e quando acabar o processamento, a próxima requisição que chegar, a anterior estará aberta. Então essa é a ideia de utilizarmos o Pool de conexões e a interface Datasource.

[07:37] Vamos só tirar esse `import java.sql.DriverManager`, que não será utilizado. Agora vamos na classe `TestaConexao` e nós vemos que não mudou nada para nós, continuamos compilando do mesmo jeito. Só que agora com a diferença que quando eu instancio uma `ConnectionFactory`, eu estou instanciando, estou criando o meu Pool de conexões.

[08:02] Depois, quando eu uso `.recuperarConexao()`, eu estou só estou pegando a conexão que está no meu Pool, então é essa a diferença, e que faz total diferença em uma aplicação que processa milhares de registros, enfim, essa mudança faz toda a diferença. Se nós executarmos essa classe `TestaConexao`, nós vamos ver que vamos ter o mesmo cenário que nós tínhamos anteriormente.

[08:31] E o motivo de trazermos esse JAR, o "mchange-commons-java-0.2.16", é que ele é quem vai *logar*, referente ao Datasource, então ele vai trazer as configurações que estão atualmente no Datasource, porque temos outras configurações que podemos fazer. Nós não vamos nos aprofundar nessa aula, mas aqui estamos falando mais ou menos isso: inicializou o Pool, e foi botando algumas informações, que depois, com o tempo, vamos entendendo.

[09:07] Na documentação vai ter sobre todas essas configurações que são possíveis fazer com o C3P0, nós não podemos nos aprofundar aqui, porque são muitas configurações, mas enfim, é basicamente para isso que ele serve. Então agora, para termos um exemplo mais real, vamos recuperar a nossa listagem do banco de dados, que vamos ver também que ela continua do mesmo jeito.

[09:35] Então nós temos agora aqueles dois registros que nós sempre estávamos usando ao longo do curso. Então, aparentemente, o nosso Pool, ele está configurado corretamente, estamos recuperando a conexão do Pool de conexões, por isso estamos conseguindo visualizar as informações do banco.

[09:54] Só que agora temos algumas maneiras de trabalhar com esse Pool, que é *setar* o tamanho do Pool, porque eu não quero deixar livre. A nossa intenção ao usar o Pool é justamente para termos um controle melhor das nossas conexões, então nas próximas aulas, vamos conseguir melhorar essas nossas configurações, porque vamos evitar os problemas de termos várias conexões abertas, enfim.

[10:23] Então hoje, nesta aula, estamos tratando sobre a implementação do Pool, porque foi algo que vimos apenas no desenho, então agora a ideia é que vamos de fato vendo os benefícios desse Pool de conexões. Então ficamos por aqui, espero que vocês tenham gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Vamos dar continuidade no assunto Pool de conexões. Até agora nós vimos o quê sobre o Pool de conexões? Nós vimos como ele funciona, para isso nós fizemos um desenho mostrando que o Pool de conexões, ele já tem um número X de conexões abertas para processar as requisições.

[00:24] Após entendermos esse desenho, nós partimos para o código e usamos o C3P0 para configurar o nosso Pool. Tem a interface Datasource agora, que é ela quem vai expor as informações do nosso Pool de conexões. Só que até agora, nós não conseguimos ver de fato o Pool de conexões funcionando. Como é que ele funciona?

[00:47] Beleza, eu vi que eu tenho um número X de conexões já abertas e que essas requisições, quando forem feitas, já vão ser processadas pelo Pool. Mas eu não vi isso funcionando. O que eu preciso fazer? Então, para conseguirmos testar o nosso Pool de conexões, a primeira coisa que teremos que fazer é ir na `ConnectionFactory`, nas configurações do Pool.

[01:11] Eu vou ter que *setar* o número máximo de conexões que eu quero permitir que sejam abertas. Então eu tenho um parâmetro, um método: `.SetMaxPoolSize()`. Eu vou definir ele como `comboPooledDataSource.setMaxPoolSize(15);`. Então quando eu instanciar a minha Connection factory, que ele carregar essas minhas informações no Pool, eu quero que ele já carregue o Pool com 15 conexões disponíveis.

[01:40] Mas como eu testo? Então eu deixei uma classe `public class TestaPoolConexoes` criada e ela só tem o main. Então vocês vão criar essa classe no projeto de vocês e para isso nós vamos instanciar a `ConnectionFactory`, que vou chamar de `connectionFactory`, e vou instancia-la: `ConnectionFactory connectionFactory = new ConnectionFactory();`.

[02:03] Quando eu instancio, mais uma vez nós vamos ver que ele já vai carregar então todas as informações do nosso Pool de conexões. Só que eu quero fazer o quê? Eu quero fazer um laço, fazer um `for (int i = 0; i < 20; i++)`, e eu quero requisitar 20 conexões. Eu quero fazer 20 requisições para o meu Pool de conexões.

[02:25] Como que esse `ConnectionFactory.recuperarConexão();` vai se comportar? Deixa só eu adicionar o `throws SQLException`. Então eu instanciei o meu `ConnectionFactory`, peguei as informações do meu Pool e agora eu estou solicitando 20 conexões. Mas eu só tenho 15 conexões disponíveis no meu Pool.

[02:48] Vamos ver como é o comportamento do Pool de conexões. Então eu vou botar aqui `System.out.println("Conexão de número: " +i);` e vou concatenar a nossa variável que está contando o número de conexões abertas no nosso Pool - com o nosso banco de dados, na verdade. Então vamos lá.

[03:16] Se eu mandar executar esse `TestaPoolConexoes`, vou dar um "Run As > Java Application", ele vai nos trazer as informações e está aqui o nosso Pool de conexões funcionando. Veja bem, eu tenho 15 conexões na minha configuração do Pool, então tenho 15 conexões disponíveis.

[03:37] No teste, eu solicitei 20 requisições, eu falei para ele me fazer 20 requisições. Como que isso vai acontecer? Mais uma vez testando. Está aqui, então ele só abriu de fato as 15 conexões com o nosso banco de dados, Você pode estar me perguntando: como é isso na vida real? Como funciona? E as outras requisições?

[04:02] As outras requisições, elas vão esperar o processamento das que já estão ocupando, digamos assim, uma das conexões do Pool que estão abertas. Então essa é aquela maneira que nós vimos, nós evitamos abrir descontroladamente várias conexões e vamos limitar essa conexão. Só que não vamos enfileirar igual nós enfileiraríamos se tivéssemos apenas uma conexão aberta, porque o processamento é muito rápido.

[04:33] Quando processar a requisição que está na primeira conexão aberta, processou, se tiver alguém chegando para ser enfileirado, quando ela já for liberada, essa que estava chegando já pode utilizar uma conexão aberta. Então seria mais ou menos esse o mecanismo, nós chegamos a ver isso no desenho.

[04:56] Se eu vir no MySQL 8.0 Command Line Client, eu já tinha feito uns testes, chamando o `show processlist;`. Só que agora é basicamente isso, se eu mandar executar o `show processlist;` de novo, eu vou ter 30 conexões, porque eu executei duas vezes o meu Pool de conexões. Então se nós contarmos, vai ter 1, 2. 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30.

[05:32] Lembrando que essa primeira da lista não é da nossa Database loja virtual e essa última, está como "starting", é o próprio MySQL 8.0 Command Line Client. Então ele abriu as 30 requisições porque nós chamamos duas vezes a nossa `TestaPoolConexoes`. Então está aqui um exemplo de como funciona o Pool de conexões.

[06:00] Esse é um teste que dá para vocês brincarem bastante aumentando o número máximo do Pool, pode colocar - temos outros parâmetros, como o número mínimo do Pool. Então, como eu já tinha falado, a documentação também da C3P0 tem explicando todos os métodos , o que cada um faz, o que cada configuração vai mudar e no seu Pool de conexões.

[06:28] Então eu recomendo que você dê uma lida nessa documentação para poder, de fato, aproveitar todos os recursos desse Pool de conexões. Hoje foi mais para mostrar mesmo como ele funciona, limitando o número de conexões, fazer um teste fazendo mais requisições que o número máximo disponível de conexões.

[06:51] Enfim, foi mais para nós brincarmos mesmo um pouco, ver o Pool de conexões funcionando, mas nada impede vocês de estudarem a documentação e implementarem do jeito que vocês gostarem na aplicação de vocês. Então é isso, aluno. Espero que vocês tenham gostado e até o próximo vídeo.

O que aprendemos?

Nesta aula, aprendemos que:

- É boa prática usar um ***pool* de conexões**
- Um *pool* de conexões administra/controla a quantidade de conexões abertas
    - Normalmente tem um mínimo e máximo de conexões
- Como existe uma interface que representa a conexão (`java.sql.Connection`), também existe uma interface que representa o *pool* de conexões (`javax.sql.DataSource`)
- **C3PO** é uma implementação Java de um *pool* de conexão

Transcrição

[00:00] Fala, aluno. Tudo bom? Dando continuidade ao nosso curso JDBC, gostaria de voltar na nossa classe `TestaInsercao` e verificarmos aqui uma questão. Hoje, na minha base de dados, eu tenho uma tabela chamada produto. Essa tabela produto, ela tem os seus atributos: o ID, nome e descrição.

[00:25] Do lado do Java, nós viemos tratando esses atributos apenas como strings, ou seja, eu não tenho uma representatividade de produto igual nós temos do lado do banco de dados. Então, por exemplo, quando eu quero fazer o *bind* dessas interrogações do `VALUES (?, ?)`, que eu quero *setar* de fato os valores dessas interrogações, eu estou recebendo nesse método `adicionarVariavel` uma `(String nome, String descricao, PreparedStatement stm)`.

[00:53] E estou *setando* essas strings no meu `setString` do meu `PreparedStatement`. Só que isso não fica muito legal, eu queria de fato ter a mesma representatividade do lado do Java que eu tenho no banco de dados. Então eu queria ter um modelo de produto. Tem uma forma de fazer isso no Java e nós vamos criar agora dessa forma.

[01:25] Dentro do "New Java Class", do "package" "br.com.alura.jdbc.modelo", eu vou criar uma classe chamada "Produto". Essa `public class Produto` vai conter exatamente o que eu tenho na minha tabela: `private Integer id;`, eu vou ter `private String nome;` e uma `private String descricao;`. Colocamos todos os atributos como privados por conta do encapsulamento da nossa classe.

[02:07] Por enquanto a classe vai ficar dessa forma porque eu não quero gerar métodos e nem nada desnecessário nesse momento. Agora eu quero fazer um teste inserindo um produto com essa classe, com a representatividade dessa classe

`Produto` e não apenas strings soltas no código. Então eu vou criar mais uma classe e ela vai se chamar `TestaInsercaoComProduto`, que nós vamos utilizar o `Produto` para inserir um produto.

[02:41] Eu já vou mandar gerar o `public static void main`, para não precisarmos ficar gerando ele. Então, primeira coisa: vamos instanciar o nosso produto. E o nosso produto vai ser uma cômoda agora no banco de dados. Vou dar um `Produto comoda = new Produto("", "");`. Só que eu já quero instanciar o meu produto passando o nome dele e a descrição, vai ser `Produto comoda = new Produto("Cômoda" , "Cômoda Vertical");`.

[03:15] Ele está reclamando, falando que eu não tenho ainda esse construtor que recebe duas strings. Vamos na nossa classe `Produto` então, "Ctrl + 3" e eu vou na opção "Generate Constructor using Fields - Choose fields to initialize and constructor from superclass to call". Então vamos criar um construtor que tenha os atributos. Não vou selecionar o ID, só o "nome" e a "descricao". Mandou gerar.

[03:44] Deu um espaço aqui no código, para ficar organizado. Agora na nossa `TestaInsercaoComProduto` está tudo certo. Pronto. Agora para começarmos a conversar com o banco de dados, eu preciso perguntar para o meu Pool de conexões, se tem uma conexão disponível para nós.

[04:05] Então eu chamo o `try()`, e estou recuperando uma connection. Vou perguntar para a nossa Connection factory se tem uma conexão disponível no Pool de conexões. Então `try(Connection connection = new ConnectionFactory().recuperarConexao())`. Vamos adicionar o `throws SQLException`, porque senão ele vai reclamar.

[04:33] Estamos querendo testar o `TestaInsercaoComProduto`, então o nosso comando vai ser o Insert. Então eu vou escrever aqui `String sql = "INSERT INTO PRODUTO (NOME, DESCRICAO) VALUES (?, ?)";`, o `VALUES` com dois atributos que ainda vão receber o seu conteúdo, o nome e a descrição do produto.

[05:05] Agora eu vou preparar o Statement, então eu vou pegar `try(PreparedStatement pstm = connection.preparedStatement(sql))`, passo o `sql`, só que eu quero, no momento da inserção, recuperar a nossa chave gerada, então vai ser `try(PreparedStatement pstm = connection.preparedStatement(sql, Statement.RETURN_GENERATED_KEYS))`.

[05:31] Agora nós vamos *setar* os atributos. São duas strings, então `pstm.setString(1, x);` no primeiro parâmetro, e o nome vai ser o `pstm.setString(1, comoda.getNome());`. Obviamente esse parâmetro não existe ainda porque nós não criamos o método `.getNome()` na nossa classe `Produto`. E também vamos ter o mesmo problema com o segundo parâmetro, que é `pstm.setString(2, comoda.getDescricao);`.

[06:05] Para o `TestaInsercaoComProduto` compilar, nós temos que ir na nossa classe `Produto` e vamos mandar criar. "Ctrl + 3" novamente. Vou mandar um "Generate Getters and Setters - Generate Getter and Setter methods for type's fields". Expando o "descricao", só vou usar o "getDescricao()". Expando o "nome" e só vou usar o "getNome()".

[06:22] A ideia, na nossa classe `Produto` é criarmos só os métodos que vão ser utilizados mesmo, para não ficar uma classe com getters e setters desnecessários. E agora vou mandar executar o `TestaInsercaoComProduto` com o `pstm.execute();`, bem padrão, como já fizemos isso em outras oportunidades.

[06:45] Agora eu quero recuperar a chave gerada. Nós vimos que para recuperar a chave gerada, nós temos o `try(ResultSet rst = )` e, para pegarmos essa chave, vai ser `try(ResultSet rst = pstm.getGeneratedKeys())`. Enquanto eu tiver um próximo, eu vou *setar* então no meu `Produto` o ID. Eu vou pegar `while(rst.next()){ , comoda.setId(rst.getInt(1)); ,` que no primeiro index é o ID.

[07:27] Obviamente não vai compilar também, porque nós não temos o `setId` em `Produto`. Então deixa eu dar um "Ctrl + 3" de novo , dei um "Ctrl + F3" sem querer. "Ctrl + 3", "Generate Getters and Setters - Generate Getter and Setter methods for type's fields", expando o ID, "setId(Integer)".

[07:49] Vamos só tirar os espaços que ele criou aqui, desnecessários. Também quero tirar esses dois espaços. Agora pegou aqui, a nossa classe `TestaInsercaoComProduto`, ela já não está mais com nenhum erro, está compilando perfeitamente. Só que agora eu quero fazer o seguinte, eu quero mostrar qual foi o produto criado.

[08:16] Então para isso, em `TestaInsercaoComProduto`, eu vou dar um `System.out.println(comoda);`. Se eu mandar executar dessa forma, ele vai me mostrar o valor da variável na memória, então ele vai me trazer aquele número esquisito e eu não

quero isso. Mas também eu não quero sair escrevendo getters e setters desnecessários, só para mostrar no nosso console o produto de forma bonita.

[08:45] Então, para isso, eu tenho uma solução: em `Produto`, eu vou sobrescrever o `public String ToString()` e para esse `return super.toString();` eu troco para `return String.format();` e vou falar `("O produto criado foi: %d, %s, %s", );`. Aqui eu estou falando que o primeiro vai ser substituído pelo ID, o segundo pelo nome e o terceiro pela descrição. Já vamos ver o resultado agora.

[09:32] Então vou passar `return String.format("O produto criado foi: %d, %s, %s", this.id, this.nome, this.descricao);`. Se mandarmos testar agora a nossa classe `TestaInsercaoComProduto`, ele vai *logar* as informações no nosso Pool de conexões, que nós vimos anteriormente e vai mostrar que o produto criado foi o de ID 110, a nossa cômoda, cômoda vertical.

[10:08] Agora nós temos, de fato, a representatividade de um produto do lado do Java também, só que agora resolvemos um problema, mas eu queria chamar a atenção de vocês para uma outra questão: toda vez que eu vou escrever as classes, os meus main, eu tenho que repetir o `try`, com o `recuperarConnection`, tenho que escrever o `String sql`, tenho que preparar um Statement.

[10:41] Enfim, eu tenho que fazer todo esse passo, seja para inserir, seja para listar. Então fica um alerta aqui. Nós já vimos sobre repetir código em várias classes, isso é um exemplo disso. Como nós podemos melhorar esse código? Talvez tenha uma forma de extrair isso para um método? Enfim, vou deixar essa questão com vocês, porque a nossa aula de agora fica por aqui.

[11:16] Espero que vocês tenham gostado agora de trabalhar com o `Produto`, que é de fato a representatividade de um produto, então agora acho que fica até mais fácil de ler o código, vai ficar mais organizado. Mas vamos deixar essa indagação e nas próximas aulas nós vamos resolver essa questão. Então, espero que você tenha gostado e vejo você no próximo vídeo.

Transcrição

[00:00] Fala, aluno. Tudo bom? Anteriormente nós vimos como utilizar uma classe de modelo para fazer persistência dos nossos dados. Só que ao realizar os testes, nós encontramos um outro problema: nós vimos que toda vez que nós precisamos inserir ou buscar informações no nosso banco de dados, nas nossas classes de teste nós estamos repetindo código.

[00:27] Então se eu quero inserir um produto, eu pego e repito uma string com a query, preparo o Statement, executo o meu SQL, enfim, fazemos uma série de repetições em várias classes. A ideia é melhorar isso. Nós já vimos que não é nunca aceitável sairmos repetindo código na nossa aplicação. Então qual seria um ponto interessante?

[01:06] Talvez, se eu estou inserindo um produto, se eu estou trabalhando com um produto, eu poderia ter uma classe onde quando eu instanciasse essa classe, eu tivesse um método salvar produto, porque eu só precisaria passar o produto, que vai ser inserido, passar uma conexão para esse método e ele faria o trabalho para mim. Então eu deixaria uma classe mais específica para as operações de banco de produto.

[01:43] Então, para testarmos isso, eu vou criar uma classe e eu vou chamar ela de "PersistenciaProduto". Essa classe, eu quero que ela tenha um construtor e que ela receba uma Connection, então `public PersistenciaProduto(Connection connection)`. Eu vou ter também um atributo, o `private Connection connection`.

[02:22] Vou pegar então `this.connection = connection;` e, como falado, eu quero um método `public void salvarProduto()`. Esse método, ele vai receber um produto, então `public void salvarProduto(Produto produto)`. Eu vou mandar importar esse primeiro `Produto`. E qual seria a lógica do meu `salvarProduto`?

[02:53] Seria essa lógica que nós utilizamos para fazer o teste na classe `TestaInsercaoComProduto`. Então toda essa `String sql = "INSERT INTO PRODUTO (NOME, DESCRICAO) VALUES (?, ?)"` eu posso extrair da classe `TestaInsercaoComProduto` e posso jogar para dentro do `PersistenciaProduto`.

[03:14] Só que antes nós trabalhávamos com um produto específico, agora não, agora eu quero receber qualquer produto, então substitui o `pstm.setString(1, comoda.getNome());` por `pstm.setString(1, produto.getNome());` e o `pstm.setString(2, comoda.getDescricao());` por `pstm.setString(2, produto.getDescricao());`.

[03:25] E o `comoda.setId(rst.getInt(1));` por `produto.setId(rst.getInt(1));`. Esse produto vai ser adicionado à nossa base de dados. O que ele está reclamando aqui? Vou adicionar o `throws SQLException`. Nosso código parou de reclamar. Aqui dentro de `try(Connection connection = newConnectionFactory().recuperarConexao())`, eu só vou precisar dar um `new PersistenciaProduto(Connection).salvarProduto(produto);`.

[03:56] No nosso caso, eu vou passar `new PersistenciaProduto(Connection), salvarProduto(comoda);`. Veja a diferença já no nosso código. Eu tenho uma classe especializada em trabalhar com a persistência das informações de produto e nas minhas classes de teste, ela fica mais sucinta, bem mais fácil de ler, de dar manutenção. Só que vamos supor que nessa situação, após eu salvar o produto, eu queira listar esse produto.

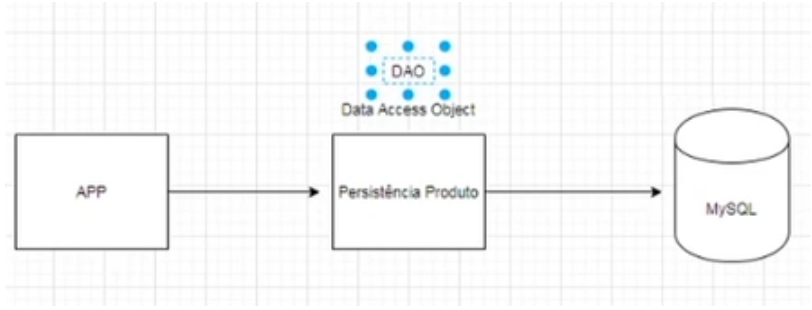
[04:36] Então eu não vou poder mais chamar ele dessa forma, com `new PersistenciaProduto(Connection), salvarProduto(comoda);` - quer dizer, vou poder, mas vai ser mais fácil então eu instanciar a minha `PresistenciaProduto persistenciaProduto = new PersistenciaProduto(connection);`, passando a minha `connection` para ser o meu construtor.

[04:59] E, no lugar de `new PersistenciaProduto(Connection), salvarProduto(comoda);`, eu uso só a minha referência, a `persistenciaProduto.salvarProduto(comoda);`. Então, se depois de salvar eu precisar listar, eu vou criar aqui o `//persistenciaProduto.listar();`. Esse comando devolve uma `Lista = //persistenciaProduto.listar();`. Enfim, esse comando não existe ainda, só estamos pensando como ficaria a nossa situação.

[05:33] Vamos ver essa classe `PersistenciaProduto`, o que ela está fazendo. No desenho - vou fazer um desenho aqui. Eu tenho o nosso banco de dados, ele é o nosso MySQL. Eu tenho a nossa aplicação. Eu tenho agora um item que se chama Persistência Produto. O que ele faz? Ele acessa os dados do nosso objeto.

[06:30] Então eu tenho um objeto produto e quem está fazendo acesso desse objeto no nosso banco de dados é essa classe Persistência Produto. Então, a minha aplicação chama a Persistência Produto, a Persistência Produto vai no MySQL e nos traz as informações. Essa Persistência Produto, ela é um Data Access Object.

[07:08] O Data Access Object é quem trabalha com as informações do nosso objeto no banco de dados. Então ele é uma DAO. Essa Persistência Produto, se ela é um Data Access Object, não faz muito sentido eu trabalhar com o nome `PersistenciaProduto`. Eu posso utilizar o DAO, produto DAO, porque seu usar Produto Data Access Object, então vai ficar muito grande o nosso nome.



[07:50] Então eu vou fazer um "Refactor > Rename" em `PersistenciaProduto` e vou utilizar "ProdutoDAO". Vou clicar em "Finish". Eu já tenho um pacote criado, vocês podem criar o de vocês, mas para ficar mais organizado, eu vou mandar o "Produto DAO", vou dar um "Refactor > Move" para esse pacote "br.com.alura.jdbc.dao". Perfeito.

[08:21] Então agora, com esse `ProdutoDAO`, tudo o que é referente a acesso ao banco de dados para trabalhar com as informações do meu objeto vão ficar nessa classe. Então, se eu quiser alterar uma informação de produto, que está no meu banco de dados, eu vou criar o método dentro de `ProdutoDAO`.

[08:50] A vantagem é que quando eu tenho um `ProdutoDAO`, eu não preciso mais especializar aqui os meus métodos para informar que eles são um produto, porque ele já está dentro de uma classe que só trabalha com produto. Então, `TestaInsercaoComProduto`, eu vou mudar `persistenciaProduto.salvarProduto(comoda);`, eu vou falar `produtoDao.salvar(comoda);`.

[09:11] Vai ficar bem bacana, porque se eu mudar de `ProdutoDAO persistenciaProduto = new ProdutoDAO(connection);` para `ProdutoDAO produtoDao = new ProdutoDAO(connection);`, quando eu for ler as minhas chamadas ao método, eu vou ver `produtoDao`, chama o seu método `.salvar();`. Então eu já estou sabendo aqui que o que o método `.salvar();`, ele é de produto, ele vai salvar um produto.

[09:43] É importante ressaltar que esse DAO, não sou eu quem estou inventando, ele é um *pattern*, então ele é o padrão da linguagem, que tudo que é sobre acesso a objeto, seja banco de dados, seja algo externo à sua aplicação, geralmente ele vai ter o sufixo DAO. É obrigatório? Não.

[10:13] Mas é aquilo: tudo o que é convenção, tudo o que é padrão, mesmo que não seja obrigatório, eu recomendo fortemente utilizar, porque vai ser assim na sua aplicação, vai ser na minha aplicação e vai ser nas aplicações corporativas espalhados pelo mundo todo. Então a ideia da DAO é exatamente isso: criamos um local onde eu vou trabalhar com a parte de persistência da minha aplicação, ela fica toda centralizada nessa DAO.

[10:46] A DAO tem por objetivo conversar com o nosso banco de dados, então onde eu vou salvar produto, eu posso chamar de qualquer main desses que eu já criei. Só que eu sempre vou instanciar a minha DAO de produto e vou passar o meu produto para ela. Dentro da DAO, esse método, o `.salvar()`, o `.listar();`, eles vão se virar, mas em um único lugar vamos ter centralizado todos eles e eles que vão executar essas querys com o nosso banco de dados.

[11:26] Então a nossa aplicação agora já começa a ficar com essa nossa camada de dados de persistência bem mais organizada, o nosso código tende a ficar mais organizado também. Agora, quando eu for fazer os próximos testes, eu não preciso mais sair criando de novo todos esses SQLs, porque ele vai estar centralizado já no mesmo lugar.

[11:54] E agora já fica utilizando a DAO, utilizando um Pool de conexões, as interfaces JDB, Datasource, já começamos a ter uma aplicação do mundo real. Quando você for trabalhar em aplicações corporativas, elas vão ter todos esses conceitos que estamos aplicando aqui e isso é muito legal. Então, aluno, eu espero que você tenha gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Dando continuidade ao nosso curso de JDBC, vamos voltar à nossa classe `TestaInsercaoComProduto` e vamos revisar o que foi feito até agora, referente à nossa camada de persistência. Nós vimos que foi necessário criar uma DAO, porque essa DAO vai ficar responsável por toda a parte de comunicar com o nosso banco de dados.

[00:27] Então é ela quem vai conter as nossas queries, ela quem vai conter os nossos Prepared Statements, porque eles estavam repetidos nas nossas classes main. Nós vimos que código repetido nunca é uma boa prática no desenvolvimento de software. Então nós chegamos no `ProdutoDAO`, e esse `ProdutoDAO` agora, ele contém um método `.salvar();`.

[00:52] Todo mundo que precisar salvar um produto, vai apenas instanciar `ProdutoDAO`, passar uma connection para o seu construtor e vai passar o produto que deseja salvar. Só que nós vimos que agora nós podemos executar mais de um comando na nossa DAO, foi inclusive em um exemplo que nós estávamos usando para verificar, que agora eu quero salvar e depois eu quero listar.

[01:22] Então por isso que até extraímos a instância da nossa DAO, porque a partir desse momento precisamos só usar referências e ir chamando os seus métodos. Eu não preciso toda vez instanciar um `ProdutoDAO`, passando a connection, faço isso uma vez só, e chamo os seus métodos através da referência. Só que agora precisamos desenvolver o nosso método `.listar()`.

[01:48] Agora, como nós temos uma DAO e nós vimos que toda parte de persistência vai ficar nessa DAO, vamos implementar o nosso método `listar()` depois do método `.salvar()`. Hoje nós temos uma classe de modelo, que representa o nosso

produto. Quando eu mando listar sem nenhum filtro, ou seja, sem nenhum where na nossa query, ele vai listar todo mundo.

[02:13] Então eu vou ter uma lista de produtos, porque se eu tiver um, ele vai ter uma lista com um produto; mas se tiver vários, vai ter uma lista com N produtos. Então o nosso método vai retornar uma lista de produtos e eu vou chamar ele de `listar()`, então fica `public List<Produto> listar()`. Eu já vou instanciar a nossa lista, que nós vamos precisar lá na frente.

[02:36] E eu vou chamar ela de `produtos()`, e vou instanciar aqui `List<Produto> produtos = new ArrayList<Produto>();`. Vou escrever o nosso SQL, que vai ser um *select* simples, ele não vai conter nenhum filtro, como já falado anteriormente, então vamos só fazer um `String sql = "SELECT ID, NOME, DESCRICAO FROM PRODUTO";`, onde temos o nome, o ID e a descrição.

[03:02] Como nós já estamos recebendo a nossa conexão pelo construtor, eu só preciso preparar, recuperar o meu Prepared Statement, e para isso eu vou usar `try(PreparedStatement pstm = connection.preparedStatement(sql));`, e passo o `(sql)`, que nós acabamos de escrever no ´String sql`. Lembrando sempre de adicionar o` throws SQLException`.

[03:29] Com o Prepared Statement em mãos, eu só preciso agora mandar executar com `pstm.execute();`. E esse código vai me trazer um resultado, isso nós já fizemos, vocês lembram bem. Esse resultado é um `try(ResultSet rst = pstm.getResultSet())`. Então enquanto eu tiver resultado, enquanto eu tiver um próximo resultado, traga para mim, com `while(rst.next())`.

[03:59] Só que agora estamos trabalhando com a nossa classe modelo. Então nada mais justo do que eu transformar esse `.getResultSet()`, que antes nós só guardávamos em strings, em integer e retornava, agora eu vou criar um produto desse `.getResultSet()`. Então, se vocês lembram bem, nós criamos um construtor em `Produto` para que quando eu quiser inserir um produto, eu já passo um nome e uma descrição.

[04:27] E quando esse construtor for inserido, ele vai me retornar a sua chave gerada. Só que agora eu tenho um cenário diferente, esse `ProdutoDAO`, ele já tem o ID, já tem o nome, a descrição, então eu vou criar um novo construtor em `Produto`. Eu posso até fazer aqui, `public Produto`, só que agora ele vai conter todas as informações.

[04:54] Então eu quero `public Produto(Integer id, String nome, String descricao)`. Vocês já vão entender porque eu estou fazendo isso. Então `this.id = id;`, `this.nome = nome;` e `this.descricao = descricao;`. Eu vou fazer isso agora porque quando eu for instanciar o meu `Produto` em `ProdutoDAO`, o que eu vou precisar fazer é o seguinte, eu vou usar esse `new Produto(id, nome, descricao)`.

[05:30] E nesse `new Produto`, eu vou fazer assim, eu vou pegar o `srt.getInt(1),`, que vai estar no primeiro Index, que é o ID, vou pegar uma `rst.getString(2)`, que é o nosso nome, que está no segundo Index, e vou pegar outra `rst.getString(3)`, que é a nossa descrição, que está no terceiro Index. Pronto, esse `new Produto(rst.getInt(1), rst.getString(2), rst.getString(3));` eu agora estou transformando ele em um produto.

[05:58] Como eu tenho que retornar a lista de produtos, o que eu vou fazer? Eu vou pegar a minha lista e vou `produtos.add(produto);`. Então o que eu estou fazendo? Eu recuperei o primeiro produto - na verdade eu recuperei a primeira linha do nosso banco de dados, transformei ela em produto e adicionei na minha lista de produtos, porque agora eu tenho que adicionar o `return produtos;`.

[06:26] Eu vou retornar essa lista de produtos, que era intenção desde o começo. Então agora eu tenho o meu método já pronto, retornando um produto, agora já utilizando a nossa classe de modelo. No nosso método `TestaInsercaoComProduto`, agora eu vou mudar o nome dele, eu vou dar um "Refactor > Rename" e vou botar "TestaInsercaoEListagemComProduto", para fazer mais sentido.

[06:57] Vou pedir para ele mudar em todo mundo que eu usei como referência. E agora eu vou fazer o seguinte, eu vou recuperar uma lista de produtos, que eu vou chamar `List<Produto> listaDeProdutos = produtoDAO.listar();`, vou chamar a nossa DAO e o método `listar();`. Para vermos se deu certo, eu vou fazer o seguinte, eu vou pegar a lista de produtos, vou usar o `stream()` para usar um `.foreach`.

[07:29] Vou botar `lp`, de lista de produtos, vou usar a *lambda* e fazer o seguinte, como nós sobrescrevemos o `toString`, eu posso já dar um `println()` no objeto, então fica `listaDeProdutos.stream().foreach(lp -> System.out.println(lp));`. Quando sobrescrevemos o `Produto`, eu coloquei `("O produto criado foi:`, o que não vai fazer muito sentido na listagem.

[08:00] Então eu vou botar `return String.format("O produto é: %d, %s, %s", this.id, this.nome, this.descricao)`. Na ´TestaInsercaoEListagemComProduto`, qual vai ser o objetivo agora? Nós vamos inserir a cômoda e depois vamos já mostrar os produtos que já existem na tabela, que são aqueles dois que viemos trabalhando desde o começo, e o novo produto, que é a cômoda.

[08:24] para verificarmos se não tem nenhum lixo, vamos usar a nossa classe `TestaRemocao`. Não tinha. Agora, se eu executar o `TestaInsecaoEListagemComProduto`, ele tem que me trazer três produtos, exatamente isso: os dois produtos que viemos utilizando desde o começo, como eu informei, e o novo, que é a nossa cômoda vertical.

[08:48] Agora nós vemos que o nosso código já ficou melhor trabalhado, digamos assim, porque eu tenho uma camada de persistência, onde eu tenho tudo o que é referente à persistência de produto, à listagem, à inserção, se eu tiver uma futura, eu posso colocar o remoção agora em `ProdutoDAO`, o remover. Enfim, tudo o que vai ser referente a produto e for trabalhar com as queries SQL, remover, o CRUD, digamos assim, vai ficar dentro de `ProdutoDAO`.

[09:23] E as nossas classes passam a ficar mais enxutas, elas ficam apenas para testar essas nossas operações com banco de dados. Eu não tenho mais queries espalhadas em todo canto, eu só preciso mesmo instanciar a nossa DAO, passar uma conexão para ela e eu posso usar quantos métodos eu quiser, desde que faça sentido, é claro.

[09:50] Então agora a nossa aplicação, ela já fica com uma cara cada vez mais de uma aplicação que você vai encontrar no mundo corporativo, nas empresas, nas grandes empresas. A preocupação é sempre essa, é melhorar o código, refatorar, não repetir código, usar padrões que já são utilizados no mundo todo, enfim é isso que viemos fazendo aqui.

[10:15] A ideia é que saiamos com esse conhecimento, com essa questão bem aprimorada, para ser automático, já pensarmos em desenvolver códigos dessa maneira, de maneiras que qualquer desenvolvedor vai ver o seu código e falar: isso eu conheço. Enfim, porque a ideia é essa: você escrever códigos de fácil manutenção. Então é isso, aluno. Espero que vocês tenham gostado e até o próximo vídeo.

08

O que aprendemos?

Nesta aula, aprendemos que:

- Para cada tabela de domínio, temos uma classe de domínio
  - Por exemplo, a tabela `produtos` tem uma classe `Produto` associada
  - Objetos dessa classe representa um registro na tabela
- Para acessar a tabela, usaremos um padrão chamado *Data Access Object* (DAO)
  - Para cada classe de domínio, existe um DAO. Por exemplo, a classe `Produto` possui um `ProdutoDao`
  - Todos os métodos JDBC relacionados com o produto devem estar encapsulados no `ProdutoDao`

Transcrição

[00:00] Fala, aluno. Tudo bom? Até agora nós estamos trabalhando, no nosso sistema, apenas com produtos. Mas estamos fazendo as operações básicas, estamos inserindo, buscando, alterando, removendo, enfim, aquele CRUD que nós viemos trabalhando ao longo do curso. Só que agora chegou uma documentação do nosso chefe, onde ele pede para nós categorizarmos esses produtos.

[00:25] Então agora o produto, além de ter um nome e uma descrição, ele quer também uma categoria. Então vamos no nosso banco de dados, no MySQL 8.0 Command Line Client, para nós vermos como está funcionando a nossa tabela. Se eu fizer `SELECT * FROM PRODUTO`, eu vou selecionar todos os produtos, nós vamos ver aqueles atributos que nós criamos na tabela e o conteúdo que nós adicionamos ao longo do curso.

[00:56] Então o que eu penso aqui? Se o nosso chefe agora quer categorizar esses produtos, eu vou poder criar então uma nova tabela, chamada categoria, e vou passar uma string quando eu for adicionar um novo produto, e eu escrevo se é um eletrônico, se é um eletrodoméstico, enfim, todas essas categorias que tem no nosso dia a dia.

[01:22] Porque, pense bem, eu, João, vou escrever a categoria de notebook Samsung, vou botar "ELETRONICO", tudo maiúsculo, sem acento. A outra desenvolvedora da equipe vai escrever, talvez, um videogame, é um eletrônico também, só que ela vai escrever "eletrônico", tudo minúsculo e com acento. Então já começamos a ter eletrônico escrito de formas diferentes, começa a duplicar as palavras no nosso banco de dados.

[01:56] Enfim, começa a ficar uma confusão. Então já vejo que essa não é uma boa abordagem. Como eu posso fazer então? Talvez eu tenha uma tabela de categorias e dê um jeito de vincular essas duas tabelas, a de produto e a de categoria. Talvez

isso faça mais sentido. Então vamos criar a nossa tabela de categoria e vamos ver como fazer para vincular essas duas tabelas.

[02:25] Então vou botar um `CREATE TABLE CATEGORIA (ID INT AUTO_INCREMENT, NOME VARCHAR (50) NOT NULL, PRIMARY KEY (ID))` , que vai ter um ID do tipo inteiro, Auto increment, vai ter um nome, que vai ser um VARCHAR de 50 caracteres, *not null* e a sua *primary key* vai ser o ID. Também vamos colocar o `Engine InnoDB` , esse, se vocês lembram, nós falamos sobre esse Engine na primeira aula.

[03:06] Agora, se eu fizer um `SELECT * FROM CATEGORIA` , não vai ter ninguém. Pensando na descrição de cada produto: Notebook Samsung, geladeira, cômoda, nós temos três categorias por enquanto. Tem um eletrônico, eletrodoméstico e móveis. Então vamos fazer logo os inserts dessas categorias.

[03:31] `INSERT INTO CATEGORIA (NOME) VALUES ('ELETRONICOS');` , a categoria de eletrônicos. `INSERT INTO CATEGORIA (NOME) VALUES ('ELETRODOMESTICOS');` e `INSERT INTO CATEGORIA (NOME) VALUES ('MOVEIS');` . Se eu fizer um `SELECT * FROM CATEGORIA` agora, nós temos que ter três categorias. Perfeito.

[04:01] Agora eu quero vincular essa tabela Categoria à essa tabela de Produto. Então eu quero, quando eu fiz um *select* de produto, eu quero que apareça o seguinte: eu tenho no ID 1 o notebook, a descrição é notebook Samsung, e eu quero a referência dessa tabela categoria, que a referência dessa tabela seja o seu ID.

[04:23] Então eu quero que tenha um número 1 na tabela produto, e eu falar: o 1, na minha tabela de categoria, é eletrônicos. Para isso, primeiro eu tenho que criar uma nova coluna na tabela de produto, chamada categoria ID. Então para isso eu vou fazer um `ALTER TABLE PRODUTO ADD COLUMN CATEGORIA_ID INT;` para adicionar uma coluna, que vai se chamar `CATEGORIA_ID` e ela vai ser do tipo inteiro.

[04:53] Criei essa coluna. Se eu fizer um `SELECT * FROM PRODUTO` agora, eu vou ter a "CATEGORIA_ID", tudo *null*, porque eu ainda não vinculei as duas tabelas. Para eu vincular essas duas tabelas, eu tenho um conceito aqui, no banco de dados, que se chama chave estrangeira. Essa chave estrangeira vai ser a que vai amarrar essas duas tabelas.

```
mysql> SELECT * FROM PRODUTO;
```

id	nome	descricao	CATEGORIA_ID
1	NOTEBOOK	NOTEBOOK SANSUNG	NULL
2	GELADEIRA	GELADEIRA AZUL	NULL
116	Cômoda	Cômoda vertical	NULL

```
3 rows in set (.00 sec)
```

[05:22] Então, como eu informei, eu quero que quando eu for adicionar um produto, eu referencie a categoria pelo seu ID. Quem faz isso é essa chave estrangeira. Para criarmos uma chave estrangeira, eu vou ter que criar também uma chave estrangeira. Nós vamos precisar fazer o seguinte, então eu tenho que fazer um `ALTER TABLE` de novo em produto.

[05:44] Eu vou adicionar uma `FOREIGN KEY` , que é a nossa chave estrangeira, falando que é a categoria ID e eu vou fazer o seguinte, referenciando, então é `REFERENCES CATEGORIA` e vou adicionar ID, então fica `ALTER TABLE PRODUTO ADD FOREIGN KEY (CATEGORIA_ID) REFERENCES CATEGORIA (ID);` .

[06:10] Então o que eu estou falando aqui é o seguinte: adicione uma chave estrangeira em Produto, que vai referenciar ao ID da categoria 16. Vamos ver se isso aqui dá certo? Beleza. Agora, se eu fizer um `SELECT * FROM PRODUTO` novamente, continua do mesmo jeito. Mas quando eu for fazer um `UPDATE PRODUTO SET CATEGORIA_ID = 1` , eu vou ter que falar o seguinte, vou falar `UPDATE PRODUTO SET CATEGORIA_ID = 1` .

[07:02] Vamos fazer assim: Update o produto *setando* a `CATEGORIA_ID` . igual a 1, que é a nossa eletrônicos. `UPDATE PRODUTO SET CATEGORIA_ID = 1 WHERE ID = 1;` , então o ID do produto é igual a 1. Então se vocês verem a tabela Produtos, onde o notebook está com o ID 1, a `CATEGORIA_ID` vai ser igual a 1, que vai ser `ELETRONICOS` .

[07:26] Se eu executar essa linha. Agora, se eu fizer um `SELECT * FROM PRODUTO` de novo, eu vou ter a `CATEGORIA_ID` do notebook agora é 1, então ele é um eletrônico, da categoria de eletrônicos. A chave estrangeira, ela é legal porque se eu tentar adicionar na tabela Produto uma referência de categoria que não existe na tabela Categoria, por exemplo o número 4, ele vai dar um erro. Então vamos fazer esse teste?

[08:02] Vou fazer o seguinte, eu vou botar `UPDATE PRODUTO SET CATEGORIA_ID = 4 WHERE ID = 2;` , na geladeira. Se eu fizer isso, ele não vai permitir porque não existe um ID 4 em Categoria. Nós não temos uma categoria que está no ID 4. Então essa é a vantagem de termos a chave estrangeira.

[08:32] Então, para terminarmos a nossa aula, vamos só *setar* então qual é a categoria dos outros produtos. A geladeira, ela vai ser `UPDATE PRODUTO SET CATEGORIA_ID = 2 WHERE ID = 2;` , de eletrodoméstico e ela também está no ID 2. Fiz esse código, mandei executar, ele executou com sucesso.

[08:55] Agora eu vou querer o `UPDATE PRODUTO SET CATEGORIA_ID = 3 WHERE ID = 116;` , então onde é a cômoda, ele vai ficar na categoria de móveis. Vou adicionar essa ID, agora quando eu fizer um `SELECT * FROM PRODUTO;` , nós vamos ter a tabela com o produto, já referenciando as suas categorias corretas.

[09:23] Então, aluno, agora nós já aumentamos a complexidade, porque nós estamos trabalhando com relacionamentos de tabelas, com chave estrangeira, enfim, alguns novos conceitos, mas que vão servir justamente para termos uma base dados normatizada, informações sem serem repetidas, nós temos uma tabela de domínio, que chama a de Categoria.

[09:48] Então aprendemos já alguns conceitos bastante interessantes e agora nós já podemos ir para a nossa aplicação e começar a construir ou a estruturar as camadas que vão representar uma categoria. Então por hoje nós ficamos aqui, espero que vocês tenham gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Voltando à nossa aplicação, agora com a tabela de categorias criadas e inclusive até com algumas categorias adicionadas, nós podemos trabalhar com aquelas informações que estão no banco de dados. A primeira coisa que eu vou fazer é criar uma classe, e eu vou botar a "TestaListagemDeCategorias".

[00:29] E eu quero que essa classe já tenha um main, porque nós vamos fazer o teste da nossa aplicação já referente às categorias. O que eu fazer, nós vamos usar aquele mesmo padrão que nós estávamos utilizando com produto, então nós vamos recuperar uma conexão, passar essa conexão no construtor de uma DAO de categorias e na DAO, vamos fazer algo semelhante ao que nós com produto também, o método de listar.

[01:06] Para isso vamos criar uma lista, botar por enquanto de categoria, ele não vai reconhecer ainda, vou botar `List<Categoria> listaDeCategorias = categoriaDAO.listar();`. Foi isso aqui mais ou menos que nós fizemos com `Produto`. Para `TestaListagemDeCatetorias`, eu quero que agora tenha `CategoriaDAO categoriaDAO = new categoriaDAO(connection);`.

[02:03] Essas duas linhas vão estar dentro de um try com recursos, então já vamos trazer as duas linhas para dentro de `try()`. E eu vou fazer o seguinte, eu quero pegar uma `try(Connection connection = new ConnectionFactory.recuperarConexao())`. Nós fizemos isso, mas por enquanto não temos nada referente à categoria, não temos um `categoriaDAO`, não temos uma classe de modelo para a categoria.

[02:49] Então vamos fazer o seguinte, eu quero criar essa classe "CategoriaDAO" e eu quero botar ela no pacote "br.com.alura.jdbc.dao". Pronto, já criamos a `CategoriaDAO`. Ele vai receber, no seu construtor, uma `public CategoriaDAO(Connection connection)` e vai existir o `this.connection = connection;`.

[03:19] Lembrando que esse `this.connection` não existe, vamos criar uma `connection`. Agora o que nós queremos é devolver, no método de listar, uma lista de categorias. Então o que nós vamos fazer é um `public List<Categoria> listar()`, `Categoria` não existe eu vou chamar `listar()`. Já vamos instanciar uma lista de categorias, que nós vamos precisar, e eu vou colocar `List<Categoria> categorias = new ArrayList<>();`.

[04:00] Esse `ArrayList<>` ainda não existe, então vou deixar ele vazio. Vamos escrever na nossa `String sql = "SELECT ID, NOME FROM CATEGORIA";`. Criei essa string, agora eu já recebi a conexão, vamos preparar o Statement, então eu vou usar o try com recursos, vou fazer o seguinte: `try(PreparedStatement pstm = connection.prepareStatement(sql))`, bem semelhante ao que nós fizemos com `Produto`.

[04:44] Então já vou mandar executar, com `pstm.excute();`. Agora eu quero fazer o seguinte, eu quero pegar o resultado disso, então vamos fazer `try(ResultSet rst = pstm.getResultSet())`. O que eu tenho que fazer agora é fazer o laço, então enquanto eu tiver um próximo, um próximo resultado, uma próxima linha no meu banco de dados, cria uma categoria e vamos criar um novo objeto do tipo categoria.

[05:33] Então `while(rst.next())` e `Categoria categoria = new Categoria;`. `Categoria` não existe ainda, vamos então criar a classe "Categoria". Vamos botar ela no pacote "br.com.alura.jdbc.modelo". Agora `Categoria` existe. O que nós vamos fazer nela é criar um `private Integer id;`, `private String nome;` e já vamos criar um construtor, que vai receber `public Categoria(Integer id, String nome)`.

[06:14] E esses atributos vão ser `this.id = id`, `this.nome = nome;`. Agora nós temos então uma Categoria. O que ele está reclamando aqui? Como não temos mais o construtor padrão, nós precisamos fazer aquele esquema que nós fizemos com `Produto`. Eu já vou pegar o `new Categoria(rst.getInt(1), rst.getString(2));`, o `getInt` na coluna 1, que é o ID e vou pegar o `getString` da coluna 2.

[07:05] O que ele está falando? Vamos adicionar o `throws SQLException`, porque senão ele não vai parar de reclamar. Com esse código em mãos, o que eu preciso fazer agora é adicionar `categorias.add(categoria);`. Então a cada laço, a cada

iteração, ele vai pegando uma informação do banco de dados, transforma em um objeto do tipo categoria e salva na lista de categorias.

[07:34] Se eu mandar adicionar um `return Categorias;`, ele vai retornar a nossa lista, bem semelhante ao que fizemos para produto. E agora, o que nós precisamos fazer em `TestaListagemDeCategoria`? Importar categoria, que nós não temos ainda, adicionar o `throws SQLException`.

[07:53] Agora, com essa lista de categorias, eu quero pegar o nome da categoria. Então vamos fazer um `listaDeCategorias.string.forEach(ct -> ct.getNome());`. Esse `getNome()` ainda não existe, então vamos criar o `getNome()` de `Categoria`, com `public String getNome()` e `return nome;`.

[08:30] Com isso, agora eu faço o seguinte, um `listaDeCategorias.stream().foreach(ct -> System.out.println(ct.getNome());` em `TestaListagemDeCategoria`. Pronto, com esse código agora, se eu mandar executar é para ele trazer as nossas categorias. Vamos ver. Temos agora a nossa categoria de eletrônicos, a nossa categoria de eletrodomésticos e a nossa categoria de móveis.

[09:06] Aparentemente tudo ok. Muito bom, aluno. A nossa aula era só para criarmos, como havíamos prometido nas últimas aulas, mais sobre essa parte que envolve categorias, era exatamente criar as nossas DAOs e a nossa classe modelo. Objetivo cumprido, agora vamos ver os próximos passos. Espero que vocês tenham gostado e até o próximo vídeo.

Transcrição

[00:00] Fala, aluno. Tudo bom? Nas últimas aulas, nós criamos as DAOs e vimos como relacionar as tabelas no nosso banco de dados. Só que agora, o que nós queremos ver é como esse relacionamento das tabelas vai se comportar na nossa aplicação. Nós já fizemos a listagem de categorias, nós vimos que está ok. Só que para mim essa informação é pouca valiosa.

[00:24] Por que eu quero apenas no meu sistema listar as categorias se não é para relacionar à alguma coisa? Até porque, como eu falei anteriormente, essa tabela de categoria é chamada de tabela de domínio, é só porque ela é vinculada à alguma coisa. Nós vinculamos no banco de dados mas não vimos ela funcionando na aplicação.

[00:46] Qual é a ideia aqui? Hoje nós já listamos as categorias. Agora eu quero saber quais produtos que estão nas suas categorias. Então, dada uma categoria, quais produtos que estão abaixo dela? Quais produtos pertencem a ela, digamos assim? Então vamos modificar o nosso código para começarmos a trabalhar com essas informações. Então vamos lá.

[01:13] Eu vou continuar listando o nome da nossa categoria, porque lá na frente vai ficar mais fácil para percebermos esse relacionamento. Então vamos lá. Agora, o que eu quero fazer? Em `TestaListagemDeCategoria.java`, eu quero fazer um for, então quando eu pegar a primeira categoria, eu quero pegar os produtos.

[01:36] Então eu tenho que fazer um `for(Produto produto : "")`, também porque eu posso receber uma lista de produto, que eu quero fazer. E entre as aspas duplas, eu vou fazer alguma coisa, vou recuperar esse produto. E o que eu quero fazer é o seguinte, quero mostrar, com `System.out.println(ct.getNome() + " - " + produto.getNome());`, o nome da categoria - vamos deixar bonito aqui com um traço e o nome do produto.

[02:14] Então já dá para entender o seguinte: peguei essa categoria. Quais produtos tem nessa categoria? Então se tiver mais de um produto na categoria, eu vou mostrar a categoria e o produto, a mesma categoria e o produto. Se não tem mais produto, ele vai pegar a próxima categoria e fazer a mesma coisa. Isso, aparentemente, para nós, vai funcionar.

[02:41] Mas o que eu vou fazer no `for(Produto produto : "")` para eu poder trabalhar com essa lista de produto, ou então com um único produto, enfim. O que eu posso fazer, nós relacionamos no nosso banco de dados, na nossa tabela de produtos, nós temos a `CATEGORIA_ID`, nós temos uma nova coluna na tabela Produto.

[03:05] Então talvez seria uma possibilidade eu procurar o produto, ou os produtos, pelo ID da categoria. Então como seria a ideia? Talvez na nossa `ProdutoDAO`, eu já passo a Connection e vou fazer o seguinte: `for(Produto produto : new ProdutoDAO(connection)`, tem um método `buscaCategoria()`, que busca por categoria. É uma opção já, já começa a ficar uma coisa bacana.

[03:35] Mas se eu estou no `ProdutoDAO`, então eu posso buscar esse produto e eu já passo a categoria para buscar o produto, então eu não preciso explicitar o buscar por categoria. Na leitura do método, já vamos conseguir entender: ele está recebendo a categoria por parâmetro e está buscando na DAO de produto, então estou buscando um produto por categoria.

[04:10] Então acho que a ideia já ficou mais válida, já ficou mais clara. Vamos criar esse método `public List<Produto> buscar(Categoria ct)`. Eu posso ter um ou mais produtos para a mesma categoria, então se eu tiver mais do que um produto, eu tenho que devolvê-lo em uma lista. Então vamos criar uma lista de produtos.

[04:31] Eu vou - eu sei que não é uma boa prática copiar códigos, mas é porque os códigos, eles são bem parecidos mesmo, então eu vou copiar o código do `public List<Produto> listar()` e vocês já vão entender qual vai ser a diferença. Ele vai pedir para eu adicionar o `throws SQLException`, normal. Agora, o que eu vou mudar?

[04:54] Em `String sql = "SELECT ID, NOME, DESCRICAO FROM PRODUTO";`, é o seguinte, invés de ele fazer só um select normal, agora eu vou filtrar a minha consulta, eu vou usar `String sql = "SELECT ID, NOME, DESCRICAO FROM PRODUTO WHERE`

`CATEGORIA_ID = ?";` , que a `CATEGORIA_ID` vai receber um parâmetro, então traga os meus produtos que estão naquela categoria.

[05:18] Para isso, eu preciso *setar* o ID dessa categoria. Ele vai ficar em `try(PreparedStatement pstm = connection.prepareStatement(sql))` . Vou fazer um `pstm.setInt(1, ct.getId())` naquele parâmetro, que se só existe um, eu vou pegar o ID da categoria. Não existe o `ct.getId()` na nossa classe de modelo ainda, deixa eu criar.

[05:41] Vou em `Categoria` , deixa eu criar ele, `public int getId()` , vou fazer um `return id;` . Agora o nosso método `ProdutoDAO` está compilando, não temos nenhum erro. Agora vamos voltar na nossa classe `TestaListagemDeCategoria` e vamos ver. Deixa eu ver o que ele está reclamando. Por mais que peguemos o `throws SQLException` , no main, ele pede para tratamos no momento de buscar por categoria, ele quer que isso seja tratado em um try catch.

[06:23] Nada que impeça, vamos fazer então essa melhoria. Agora o nosso código está compilando, se eu mandar buscar, eu vou verificar aqui os nossos retornos. Olha só, nós já tínhamos listado as nossas categorias, então basicamente o que ele faz, aqui ele listou e guardou em uma lista, a `listaDeCategoria` , agora ele itera sobre essa lista. Então vamos lá.

[06:58] Na categoria Eletrônicos, eu tenho o notebook e o videogame, na Eletrodomésticos eu tenho a geladeira e em Móveis eu tenho a cômoda. Parece que conseguimos vincular os nossos produtos às nossas categorias. Mas estamos utilizando duas queries, digamos assim, dois serviços de busca na mesma chamada, em `TestaListagemComCategoria` , na linha `List<Categoria> listaDeCategoria = categoriaDAO.listar();` .

[07:35] Vamos ver qual é o impacto disso, vamos colocar em `CategoriaDAO` , em `public List<Categoria> listar() throws SQLException` , um `System.out.println("Executando a query de listar categoria");` . Coloquei esse comando, e nessa query, nesse serviço de `ProdutoDAO` , o `public List<produto> buscar(Categoria ct)` , eu vou fazer um `System.out.println("Executando a query de buscar produto por categoria");` .

[08:20] Eu falei serviço, mas são as nossas DAOs, só para ficar claro. Então agora que botamos esses `System.out.println()` naqueles dois métodos das nossas DAOs, vamos executar de novo o código e vamos ver uma situação. Veja bem, foi

executada a query de listar categorias, essa query foi uma vez e guardou em uma lista. E eu vou iterar a lista de categorias justamente para eu poder vincular os meus produtos àquela categoria.

[09:02] Então quando ele itera na lista de categoria e pega Eletrônicos, então ele vai e busca os produtos por categoria Eletrônicos. Ele trouxe o notebook e o video game. Iterou na lista de categoria, ele vai mais uma vez no banco, porque a nossa categoria mudou, é Eletrodomésticos, eu tenho um novo produto. E a mesma coisa para Móveis.

[09:32] Então em uma ação relativamente simples, que eu só quero relacionar duas tabelas, então os Produtos com as Categorias, nós vamos no banco de dados 1, 2, 3, 4 vezes. Isso porque temos poucos produtos e poucas categorias. Pense em N produtos e N categoria então pense na confusão que ia ficar e o quão pouco performático é em relação à nossa aplicação, em relação à comunicação com o banco de dados. Então já vimos que esse tipo de listagem não é uma boa prática.

[10:20] Inclusive isso tem um nome, que chama queries N+1, porque nós temos uma query base, que é a `categoriaDAO.listar()`, que ele vai uma vez e busca todo mundo. Só que através dessa base, eu executo N queries para trazer as informações que eu preciso que no caso é de produto. Então, na primeira categoria, eu trago N produtos. Eu vou ao banco mais uma vez e trago N produtos, vou no banco mais uma vez e trago mais N produtos.

[10:53] Isso referenciando a categoria. Então isso não é uma boa prática e tem que melhorar, porque, se não, tende a crescer e isso vai gerar um prejuízo futuramente. Como já estamos trabalhando a um certo tempo nessa aula, não vamos entrar no Refactory nesse momento, só quero que vocês fiquem na cabeça como podemos melhorar o nosso código, o que podemos fazer para melhorar e já analisar realmente essa questão da performance.

[11:35] Talvez vocês possam brincar um pouco, adicionar mais categorias, adicionar mais produtos e ver quantas vezes vamos ter que ir no nosso banco de dados para trazer essas informações. Então, como falado, no próximo vídeo nós vamos resolver essa situação e por hoje nós vamos ficando por aqui. Então, aluno, espero que vocês tenham gostado e até a próxima aula.

Transcrição

[00:00] Fala, aluno. Tudo bom? Nas aulas anteriores, nós vimos como trabalhar com relacionamento entre duas ou mais tabelas. Nós criamos a chave estrangeira e tentamos trazer isso para o lado da nossa aplicação. A abordagem que utilizamos, nós vimos que não foi muito boa, porque nós listamos as categorias e depois nós procuramos os produtos por essas categorias.

[00:24] Com isso, nós vimos que estávamos fazendo muitas requisições para o banco de dados e isso poderia ter um problema de performance futuramente. Então vamos ter que melhorar o nosso código. Primeiro de tudo, eu não quero mais listar todas as categorias para depois buscar os produtos, eu queria, de alguma forma, trazer já as categorias com os produtos, então eu não precisaria mais fazer `new ProdutoDAO(connection).buscar(ct)`.

[00:52] Então, para isso, vamos comentar esse trecho de código e vamos pensar em uma forma de ter um método que vai listar as categorias já com os produtos. Então vou mandar criar esse `listarComProdutos()` na nossa `CategoriaDAO`. Criou, ele vai continuar nos retornando uma lista de categorias, então `public List<Categoria> listarComProdutos)` e um `return null;`. Deixa eu botar o `throws SQLException`, que vamos precisar.

[01:22] Eu vou usar como exemplo esse código do `listar()` categorias, que ele não vai ficar igual, mas nós vamos utilizar algumas coisas dele. Motivação: fazer um *select* em Categoria e já trazer os nossos produtos. Para a nossa sorte, nós já temos algo no SQL que faz isso para nós, ele é chamado de Inner join. O Inner join, para quem não conhece, já vai entender como funciona mais ou menos essa questão do Inner join.

[01:59] O que importa é que para eu usar o Inner join, eu preciso colocar um *alias* na minha categoria, preciso botar o Inner join, vou botar esse Inner join com produto, que vai ter o *alias* P e tenho que fazer o seguinte - nós já vamos entender essas palavras-chaves. Eu quero os produtos, na verdade eu quero trazer os produtos que estão relacionados com categorias.

[02:38] Isso eu vou representar dessa maneira: `String sql = "SELECT C.ID, C.NOME, P.ID, P.DESCRICAO FROM CATEGORIA C INNER JOIN" + "PRODUTO P ON C.ID = P.CATEGORIA_ID";` . Como nós vamos trabalhar com essa linha? Eu quero trazer o ID da categoria, o nome da categoria, o ID produto, o nome do produto e a descrição do produto.

[03:13] Basicamente eu estou falando o seguinte: traz as informações com ID e nome da categoria, ID e nome e descrição do produto onde existir vínculo entre produto e categoria. Pode não ficar claro aqui agora, olhando a query, mas vamos executar isso no nosso banco de dados. Vou em MySQL 8.0 Command Line Client. Se eu fizer um `SELECT *FROM CATEGORIA;` , eu tenho eletrodomésticos, eletrônicos e outros.

[03:48] `SELECT * FROM PRODUTO` , vamos ver, eu tenho quatro produtos também, eu tenho o notebook, a geladeira, a cômoda e o videogame. A categoria deles, você pode ver, eu estou usando a categoria de eletrônicos, de eletrodoméstico e móveis. A categoria outros não está vinculada, não tem nenhum produto que pertence à categoria outros.

[04:14] Quando executo aquela query com Inner join, que vocês já vão entender, `SELECT * FROM CATEGORIA C INNER JOIN PRODUTO P ON C.ID = P.CATEGORIA_ID` , eu vou trazer só os que têm vínculo realmente. Você vê, eu não tenho nenhum produto que esteja vinculado à categoria de outros, então ela nem aparece para mim.

[04:53] Essa é a ideia do Inner join, trazer só os que estão juntos, os que estão vinculados entre categorias e produto. Então essa é a ideia da nossa query, essa é a ideia do nosso Inner join. Com isso, eu já consigo melhorar a nossa chamada na nossa aplicação. Vou dar um "Run As > Java Application", não é para ter nenhum problema. Vamos ver. Ele trouxe as categorias, está listando apenas uma vez.

[05:26] Mas esse já era um cenário que era esperado. Quando listávamos apenas categorias, nós só íamos uma vez no banco de dados, então ainda não avançamos tanto. Só que agora eu vejo que ele trouxe eletrônicos duas vezes, eletrodomésticos e

móveis. A motivação disso é o que nós vimos, eu tenho dois produtos que estão na categoria de eletrônicos, só que eu não quero trazer eletrônicos duas vezes, para mim só basta uma vez.

[05:57] Vocês vão ver que quando eu trago o resultado do nosso banco de dados, ele está instanciando a mesma categoria duas vezes, então é eletrônicos e eletrônicos. Isso não faz muito sentido, eu só quero instanciar a categoria quando elas forem diferentes. Para resolver isso, como vamos fazer? Eu vou criar uma referência em `Categoria`, e vai ser `Categoria = ultima = null;`. O que eu vou fazer?

[06:26] Eu vou sempre vincular a `ultima = categoria`. Na verdade, a `ultima` sempre vai receber uma categoria. Vocês já vão entender o porquê. Para eu instanciar uma categoria, eu quero que só ocorra quando a última for igual a nulo porque significa que eu ainda não instanciei nenhuma categoria.

[06:49] Se eu estou atribuindo categoria à `ultima`, toda vez, `if(ultima == null)` é porque não teve atribuição, então eu quero que ele instancie a categoria ou `if(ultima == null || !ultima.getNome().equals(andObject))`, ou se o nome da última seja diferente, e é por isso que eu vou negar, que o nome que vem do banco de dados.

[07:17] Então eu pego o nome da `ultima` - lembrando que a `ultima` sempre vai ter a atribuição da categoria. Se o nome da `ultima` for diferente do que vem do banco de dados, significa que o que está vindo do banco de dados é outra categoria, então eu vou instanciar essa categoria. Então esse trecho de código fica dentro pode ficar dentro do `if` e acho que a nossa lógica já fica 100% e eu não vou ter mais essa repetição de eletrônicos.

[07:44] Eu não estou instanciando eletrônico duas vezes. Vamos testar? Vou rodar o programa. A nossa lógica deu certo e agora está rodando 100%. Só que agora você deve estar me perguntando: categorias ok, e produto? Produto está aqui, com o nosso Inner join, ele traz produto também, então nós temos que criar o produto.

[08:08] Para isso eu vou instanciar um `Produto produto`, ele vai receber no construtor. Se o Index 1 é o ID da categoria e o Index 2 é o nome da categoria, o 3 vai ser o ID de produto, o 4 vai ser o nome de produto e o 5 vai ser a descrição do produto, então fica `= new Produto(rst.getInt(3), rst.getString(4), rst.getString(5));`.

[08:45] Isso nós não estamos inventando, é simplesmente a ordem do nosso resultado da nossa query. Então é 1, 2, 3, 4 e 5. Coluna descrição é o 5. Quando eu criei esse `Produto produto`, eu preciso vincular ele à categoria. Como eu vou fazer isso? Faz sentido para vocês quando eu vou em Categoria, eu falo que uma categoria tem vários produtos?

[09:10] Faz sentido, tanto é que é verdade, na categoria eletrônicos eu tenho dois produtos, posso ter três produtos. Então ele vai ser uma `private List<Produto> produtos = new ArrayList<Produto>();` na classe `Categoria`. Então agora eu tenho `Categoria` recebendo uma lista de produtos. Aqui já começa a desenhar a ideia.

[09:36] Eu tenho um produto, eu preciso adicionar uma lista de produtos. Como eu vou fazer? Eu chamo um `ultima.adicionar(produto);`. Eu não tenho esse `adicionar`, mas nós criamos, não tem problema. O `adicionar`, em `Categoria`, vai fazer o seguinte, `public void adicionar(Produto produto)`, ele vai pegar a lista de produtos e vai adicionar produto, `produtos.add(produto);`, quantos produtos forem.

[10:03] E agora eu tenho, na minha `CategoriaDAO`, um método `adicionar()` e a única coisa que ele está fazendo é criar uma categoria, ele vai criar só a categoria quando a `ultima` foi igual a nulo, então ele não instanciou nenhuma categoria ainda, ou quando for uma categoria nova, quando for a mesma categoria, eles não vão instanciar, mas o produto, ele sempre vai adicionar naquela categoria.

[10:34] Acho que a lógica ficou bem bacana, temos agora a criação da Categoria e o vínculo aos seus produtos. Vamos testar. Quando eu quero testar esse código, eu já posso descomentar esse `try` porque nós vamos precisar dele novamente. Só que com a diferença que eu não vou precisar mais chamar o método de `buscar(ct)` no `ProdutoDAO`.

[11:02] Porque agora basta eu ter um `for(Produto produto : ct.getProdutos())`, que é uma lista, que nós vamos ver o resultado. Não tenho o `getProdutos()` ainda, não tem problema. Vamos em `Categoria`, escrevo o `public List<Produto> getProdutos()`, e já mando ele gerar `return produtos;`. Agora o `getProdutos()` já está funcionando, já está compilando.

[11:25] Como não tenho mais aquele buscar, passando a categoria, eu não preciso do `catch (SQLException e)`, então eu só preciso mesmo do `for` e é bom que o nosso código já fica até mais bacana, mais sucinto, fica menor. Deixa eu tirar os

`imports` que não estão sendo utilizados.

[11:48] Se eu mandar executar esse código, nós vamos ver que o resultado dele é o mesmo que nós tínhamos quando estávamos buscando os produtos com outra chamada, uma outra query. Agora nós temos que o principal objetivo é esse, nós temos a execução de apenas uma única query e vinculando certo, eu tenho eletrônicos, que tem notebook e vídeo game, eletrodomésticos, geladeira e móveis, cômoda.

[12:23] O outros nem aparece, porque eu não tenho nenhum produto vinculado àquela categoria. Então a ideia é exatamente o nosso código, ele ficar mais sucinto, ficar mais performático ao banco de dados e agora não tenho mais o que fazer, o nosso objetivo era esse mesmo. É isso, aluno, espero que vocês tenham gostado e até a próxima aula.

O que aprendemos?

Nesta aula, aprendemos:

- Que quando temos um relacionamento, é preciso ter cuidado para não cair no problema de *queries* `N + 1`
  - `N + 1` significa executar uma *query* e mais uma nova *query* (`N`) para cada relacionamento
  - *Queries* `N + 1` podem gerar um problema no desempenho
  - *Queries* `N + 1` podem ser evitadas através de *joins* no SQL
- A criar a nossa própria camada de persistência

Transcrição

[00:00] Olá, aluno. Tudo bom? Anteriormente nós vimos como resolver um problema que pode agravar bastante o nosso banco de dados, que é a questão das queries N+1. Query N+1 é quando temos uma query base e a partir dessa base, nós temos que ir N vezes no banco de dados para recuperar algo com essa query base. No nosso caso, é a `Categoria`.

[00:28] Nós recuperamos as categorias do banco, e a cada categoria, nós tínhamos que ir no banco de novo para recuperar os produtos de determinada categoria. Nós vimos que isso, dependendo do fluxo que tivermos, de requisições e a quantidade de produtos que nós tivermos na base, pode ser bem prejudicial ao banco.

[00:47] Então nós resolvemos essa questão criando uma `CategoriaDAO`, um método `listarComProduto()`, onde fazemos o Inner join e agora só precisamos ir no banco uma vez e ele já traz as categorias com os produtos. Chegando nesse ponto, já passamos por vários recursos do JDBC.

[01:05] Então nós vimos com criar o nosso Datasource, nós vimos como podemos fazer as operações com o banco de dados a partir da nossa aplicação, então fazendo select, insert, update, então tivemos bastante conteúdo ao longo do curso. Só que, geralmente, a nossa aplicação, ela vai ser acessada, essas operações, elas vão ser feitas a partir de algum lugar, uma *view*.

[01:31] O nosso usuário, ele vai entrar em algum lugar, vai ter uma tela onde ele vai poder cadastrar o produto, onde ele vai poder listar o produto. Dificilmente você vai ver usuários - dificilmente não, acho que nem existe o usuário entrando no código e mandando executar igual nós costumávamos fazer no nosso main.

[01:53] Então, por exemplo, a `TestaListagem`, nós chegávamos, mandávamos executar e ele retornava os nossos produtos na base de dados. Não faz sentido isso quando estamos pensando para o usuário. Então a ideia agora é que tenhamos um local onde o usuário, ele pode fazer as requisições de inserção, de alteração, e no banco de dados, no nosso *back-end*, que é a nossa aplicação, recupera essas informações e conversa com o banco de dados.

[02:21] Algo semelhante a "Run As > Java Application". Então vocês podem não estar entendendo agora, pois estamos em um método main, selecionando e mandando executar, e agora aparece uma tela? Mas a motivação dessa nossa aula vai ser exatamente chegarmos nesse objetivo, que é criar a nossa própria tela. Então, como seria?

[02:45] Eu tenho a tela de Produtos, onde posso, se eu errar os produtos, eu posso mandar limpar. Se eu quiser inserir um produto, por exemplo, "Microfone", "Microfone para computador", eu seleciono a categoria, então já estão bem mais interativas as minhas categorias. Então categoria eletrônicos, mando salvar.

[03:08] Está vendo? Tudo agora está interativo. O nosso usuário, agora, ele tem uma *view* aqui, ele tem uma tela para ele poder interagir, tudo auto explicativo, então onde ele bota o nome do produto, onde ele bota a descrição do produto. Se eu quiser editar um produto, esse celular, no estoque o celular sem câmera acabou, então eu só vou precisar trocar para colocar "Celular com câmera".

[03:31] Clico no botão "Alterar" e tudo isso é alterado no banco de dados. Podemos ver o resultado fazendo um select. Se eu vier no MySQL 8.0 Command Line Client e fizer um `SELECT * FROM PRODUTO`, nós vamos ver o microfone que acabou de ser inserido, já está constando no nosso banco de dados.

[03:49] Essa aplicação, o que ela está fazendo por trás é transparente para o usuário, ele não tem que ter ciência disso. Nós que temos que, se tiver algum problema, mexer no nosso *back-end*, mas o usuário só interage com essa tela, dando os comandos que ele quer fazer. A ideia não é começarmos o desenvolvimento dessa tela agora, é só entendermos onde queremos chegar.

[04:10] Nas próximas aulas, vamos fazer o passo a passo de como chegar nessa tela, fazendo todas as iterações com o banco de dados. Faltou mostrar para vocês, nessa tela, eu também consigo excluir um produto com o botão "Excluir". O item foi excluído com sucesso. Se formos no MySQL 8.0 Command Line Client, na base de dados, agora só vamos ter três produtos.

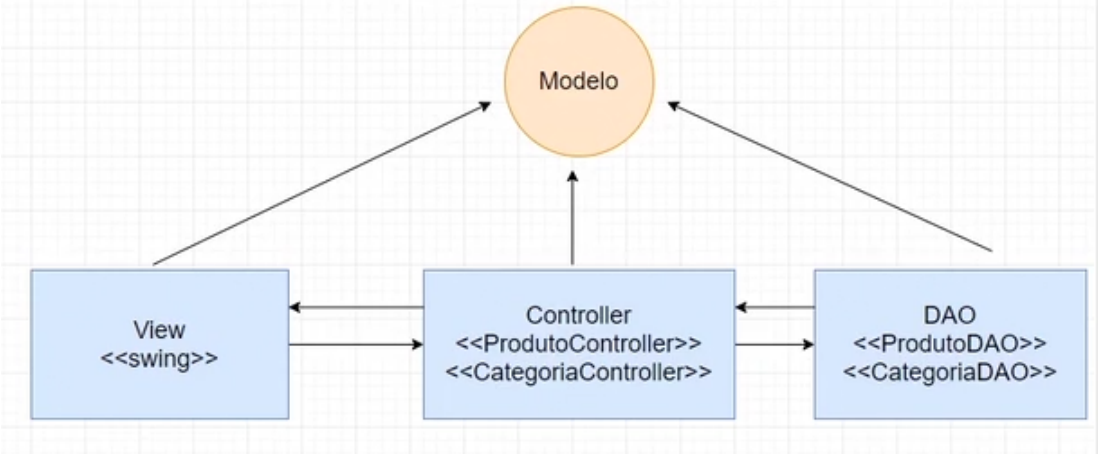
[04:31] Então você vê que realmente a nossa tela está fazendo requisição para o nosso *back-end*, que por sua vez conversa com o banco de dados. Demos a introdução de qual é o nosso objetivo, agora vamos para o próximo vídeo, que vamos entender como chegamos a essa tela. Não deixem de assistir o próximo vídeo. Valeu.

Transcrição

[00:00] Olá, aluno. Tudo bom? Na aula passada, nós conhecemos a maneira como os nossos usuários vão fazer agora a requisição para o nosso *back-end*, a nossa aplicação. Nós vimos que será construída uma tela, onde o usuário poderá preencher campo para ser inserir, poderá excluir dados, poderá alterar produtos da nossa base de dados sem se preocupar como o código faz isso.

[00:25] É o dia a dia, o usuário, ele não vai acessar o código. Nós fizemos o curso para entendermos como funciona, mas agora expomos para o usuário a partir de uma *view*. E nós vimos então onde queremos chegar. Antes de partir para o desenvolvimento, precisamos entender um pouco sobre camadas, porque não é simplesmente a nossa *view* chegando na nossa DAO, fazendo a operação e pronto, acabou.

[00:54] Nós temos uma separação de camadas para que o nosso código, ele fique mais correto, que fique mais fácil de dar manutenção nesse código. Então, para isso, eu trouxe um desenho, onde nós vamos ver exatamente o que nós estamos fazendo. Então eu tenho uma *view*, aqui nós estamos utilizando o Swing, que é do próprio Java.



[01:17] Aqui eu abro um parênteses já para falar para vocês que o Swing, ele serve para aplicações Desktop. Só que hoje em dia, dificilmente vocês verão o desenvolvimento, começar um projeto para desenvolver para Desktop. Vamos ver para mobile, para web. Desktop já é algo mais antigo.

[01:38] Mas antes de passarmos para o próximo passo, que é conhecer o desenvolvimento web, que inclusive é até a nossa próxima formação, precisamos ter um conhecimento dessas camadas, de como funciona a comunicação entre as camadas e isso podemos fazer com o Swing, que vocês vão aproveitar esse conhecimento quando vocês forem para os próximos cursos, para o desenvolvimento web.

[02:04] Pensando nisso, com o Swing sendo o nosso *front-end*, a nossa *view*, temos que fazer a requisição para a nossa aplicação, que automaticamente vai no banco de dados para fazer alguma operação com o banco de dados. Só que a nossa *view*, ela não conecta diretamente com a nossa DAO.

[02:24] Nós não podemos atrelar essas duas pontas porque a *view*, ela serviria apenas para receber requisição e não deveria, pelo menos não deveria, ter lógicas, por exemplo, de abertura de conexão, porque a nossa DAO, ela precisa, no momento em que eu instancio o meu `ProdutoDAO`, a minha `CategoriaDAO`, eu preciso passar a minha conexão.

[02:49] Então eu não posso, ou então eu não deveria - não posso, não, não deveria - fazer isso na nossa *view*. Então, para isso, eu crio um controlador. Esse controlador é quem vai receber a requisição da minha *view* e vai saber para onde tem que

enviar para a nossa DAO. Ele fornece uma conexão para a nossa DAO. Quando a DAO processa a informação, devolve para a Controller, a Controller me devolve essa informação e a *view* apenas mostra.

[03:19] Então eu tenho agora um conceito de Controller, que é de fato um controlador. Nós vamos ver que tudo o que é trafegado entre essas caixas, entre as nossas camadas, é o nosso Modelo, que é o nosso `Categoria.java`, que é o nosso `Produto.java`, que tem os nossos atributos, que tem os nossos getters, os nossos setters.

[03:41] Então esse Controller, ele vai servir também para ser trafegado entre essas camadas. Nós já tínhamos as nossas DAOs, agora nós já vimos que nós temos a *view*, o que nós precisamos fazer é implementar a nossa Controller e, especialmente essa comunicação entre as camadas. É isso que é o nosso objetivo e é isso que vamos desenvolver nessa sessão do nosso curso.

[04:10] Então a ideia aqui era trazer para vocês o conhecimento dessas camadas, o porquê agora eu tenho uma Controller, o que trafega entre essas camadas e agora, o objetivo é fazermos de fato isso no código e botar a mão na massa, e ver a tela funcionando, mas agora com a nossa implementação. Então não deixem de assistir a próxima aula e vamos botar a mão na massa. Valeu.

Transcrição

[00:00] Olá, aluno. Tudo bom? Agora que nós vimos o funcionamento da nossa tela interagindo com o nosso *back-end* e vimos que para o desenvolvimento da nossa aplicação ficar com as boas práticas necessárias, nós vimos que precisamos dividir o nosso projeto em camadas.

[00:21] Agora chegou a hora de darmos o próximo passo, que é de fato ver como vai funcionar esse projeto e botar a mão na massa para conseguirmos desenvolver esse código da melhor maneira. O projeto que nós utilizamos até agora, que é o "loja-virtual-repository", nós não vamos utilizar nessas aulas, então vocês podem dar um close na IDE de vocês, porque nós vamos importar um novo projeto, que está disponível para download na plataforma.

[00:50] No meu caso, ele já está importado, mas para você importarem quando vocês salvarem no diretório de preferência de vocês, vocês vão no menu "File > Import", selecione que você quer adicionar um novo projeto existente ao seu workspace, ele fica em "General > Existing Projects into Workspace". Clique em "Next".

[01:10] Clique em "Browse..." para selecionar o diretório onde vocês salvaram o projeto de vocês. O meu está aqui, em "eclipse-workspace". Tenho o projeto "jdbc-aula-8-projeto-completo", seleciono agora a pasta "loja-virtual-view-repository" e seleciono essa pasta. No meu caso, eu não posso selecionar o projeto, porque ele já está importado.

[01:36] O projeto de vocês estará selecionável, é só selecionar e clicar no botão "Finish" e vocês vão ver essa estrutura de projeto. Então aqui nós começamos a ver alguma diferença em relação ao nosso outro projeto. Primeiro que já não temos mais aquele tanto de main, que usamos ao longo do curso para ir testando as operações com o banco de dados, nós temos só o `TestaOperacaoComView`, que é exatamente o main que vai chamar a nossa tela.

[02:00] E aqui nós já vamos ver. Nós temos a nossa Controller, que é o controlador, que recebe a requisição da tela, encaminha para a DAO correta. Quando a DAO processa a informação, ela devolve para a Controller, que devolve para a *view*. Então é esse desenvolvimento em camadas, essa chamada em camadas que nós vimos no desenho e que fica bem representado pelos pacotes.

[02:25] As DAOs, nós já tínhamos, então não é nada novo, nós só vamos passar por elas rápido aqui, porque nós vamos ver que agora, por exemplo, em `ProdutoDAO`, eu coloquei no nosso projeto novos métodos para a nossa tela ficar com todas as operações, pelo menos as operações básicas, que são o select, o insert, o update e o delete.

[02:52] Então eu tenho agora um método `salvar`, eu tenho o `salvarComCategoria`, porque quando nós salvávamos, ele não estava salvando o ID da categoria na tabela de produto. Como agora, quando formos inserir através da nossa tela, vai ter aquela ComboBox de categorias, então agora eu tenho um novo método aqui, que é o que vamos chamar da tela, que vai inserir também o *ID da categoria.

[03:20] Então vai ficar mais conciso, vai ficar mais real o nosso exemplo. E temos o método de listar, de buscar, que nós já tínhamos também, o de deletar, alterar, tudo isso nós já tínhamos feito, só que agora eu coloquei tudo na nossa DAO. A `CategoriaDAO` não teve mudança, nós não vamos usar os métodos de categoria, vamos utilizar só o listar, que retorna todas as categorias para montarmos a nossa lista de categorias.

[03:49] Então a DAO, a única coisa que ela teve de diferente foi isso. A nossa `ConnectionFactory` continua aqui, porque nós vamos precisar ainda pegar uma conexão para interagir com o banco de dados. Nossas classes de modelo também continuam aqui, nós vimos que elas são essenciais nessa estrutura de *view*, Controller e DAO, porque são os objetos que são trafegados entre as camadas, são essas nossas classes de modelo.

[04:16] Chegamos enfim à nossa *view*. Eu vou passar com vocês, para vocês entenderem mais ou menos o que foi feito aqui na tela. Só que eu quero chamar a atenção para um detalhe aqui: nós utilizamos o Swing para criar essa nossa tela. O Swing, ele é um recurso do próprio Java, mas ele é para desenvolvimento em desktop, ou seja, para aquelas telas que nós vamos subir na nossa própria máquina.

[04:44] Hoje em dia não é mais tão utilizado, dificilmente vocês vão ver projetos que estão iniciando agora começando como um projeto desktop. Mas vamos ver projetos web, nós vamos ver projetos mobile, só que essas outras tecnologias, elas exigem recursos a mais. Então, por exemplo, se fossemos pensar no desenvolvimento web, nós vamos pensar em HTML, CSS, JavaScript.

[05:14] Nós podemos pensar, hoje, em aplicações *front-end* desacopladas do *back-end*. Então nós temos outro conhecimento que nós precisamos ter para desenvolver na web. Dessa forma, com o Swing, é interessante porque tudo o que aprendermos aqui, sobre a requisição para uma *view*, passando pelas camadas, nós vamos utilizar no desenvolvimento *web* e no desenvolvimento mobile.

[05:42] Inclusive nós temos outras formações que vocês podem fazer, quem quer ser especialista em desenvolvimento mobile, nós vamos ter as próximas formações, que vamos aproveitar esse conhecimento aqui e vamos aprofundar ainda mais o conhecimento para desenvolver na web.

[06:00] Então foi basicamente essa a nossa ideia aqui, trazer um pouco já desse conhecimento para vocês, mas indico para a próxima formação para ter um conhecimento 100% de como funciona todo esse mundo da web. Voltando para o nosso exemplo, então eu tenho o `ProdutoCategoriaFrame`, que vai ser aquela nossa tela.

[06:19] O Swing, ele vai usar objetos Java para representar os objetos da nossa tela, então pode ver que eu tenho um `JLabel`, `JTextField`. Então Label é o nome que fica em cima da caixa de texto, que no nosso caso é o nome do produto, a descrição do produto. O `JTextField` é exatamente onde escrevemos qual é o nome do produto, qual é a descrição do produto.

[06:41] Nisso nós vamos ter uma classe, um recurso do Java para cada recurso da tela: `JComboBox`, o `JButton`, o `JTable`. E como funciona? Dentro do construtor da nossa `ProdutoCategoriaFrame` que nós vamos configurar toda a tela. Então, por exemplo, esse `super("Produtos");` é o que vai ficar na barra superior quando subirmos a nossa aplicação.

[07:04] Então nós vamos ver que vai ter Produtos na parte de cima da tela, isso é só um detalhe. E todas as configurações, eu instancio as minhas Controllers, eu tenho até um atributo `private ProdutoController produtoController;`, que nós vamos

utilizar lá embaixo, vocês já vão ver os nossos métodos.

[07:22] Temos `labelNome`, então o nome do produto. Aqui eu estou configurando as minhas Labels, a posição das minhas Labels, qual vai ser a cor da minha Label. E aqui eu adiciono na minha tela, que é o meu `container`. Então a partir desse ponto eu tenho isso inserido na minha tela. Isso vai servir para todos os recursos, então eu tenho `textoNome`, `textoDescricao`.

[07:45] A única coisa que eu tenho, no `comboCategoria`, como ele é um `ComboBox` e ele lista as minhas categorias, eu tenho que fazer um `forEach` nesse método `listarCategoria`, que está chamando a nossa Controller e que vai nos retornar todas as categorias e vai adicionar na `ComboCategoria`.

[08:05] Quando eu selecionar a categoria, ele, de alguma forma, que nós vamos ver ainda, vai pegar o ID desse item e vai salvar o produto na minha tela com o ID da categoria que eu selecionei. De resto aqui, a mesma coisa, só configuração da Label, do botão, o tamanho do botão, a posição, enfim. Então no nosso construtor nós fazemos isso, além de configurar os `ActionListener` dos botões.

[08:32] Então, quando eu clico no `botaoSalvar`, ele vai chamar um método `salvar();`, vai `limparTabela();` e vai `preencherTabela();`, porque agora eu consigo mostrar esse novo recurso salvo no nosso banco de dados. A mesma coisa ele vai fazer, por exemplo, no `deletar();`, ele deleta, ele `limparTabela();` e ele `preencherTabela();`. Nós já vamos ver esse comportamento com detalhes.

[08:52] E o `botaoEditar`, o `botaoLimpar`, que apenas limpa os nosso Text fields. Aqui nós temos as nossas regras de *front-end*, de *view*. Ou seja, as regras da nossa tela. Como são essas regras? Por exemplo, eu faço validações na hora de salvar, por exemplo. Primeiro eu preciso verificar, na hora de salvar, se o meu nome e descrição, os meus Text fields, eles estão diferentes de vazio.

[09:23] Se estiverem, eu pego o produto, pego a categoria, pego todas as informações, chamo a minha `controller` e salvo. E faço um alerta, falando que que o produto foi salvo com sucesso. Se isso não for verdade, se os campos estiverem vazios, eu

dou um `JOptionPane`, que é o recurso para fazer um alerta na nossa tela, falando que o nome e a descrição devem ser informados.

[09:48] Então aqui são as nossas regras de *view*, por isso que eu não quis misturar e chamar as nossas DAO diretamente daqui, porque senão teríamos que passar conexão a partir da nossa *view* para a nossa DAO, e não faz muito sentido. Por isso o desenvolvimento em camadas, eu passo essa responsabilidade para a minha Controller, aqui são só as regras de *view*.

[10:09] Por enquanto, como nós fazemos na `TestaOperaçãoComView`? Eu tenho um main, que vai chamar a nossa classe para mostrar a tela. Por enquanto, se eu mandar executar esse código, nós vamos ver que está diferente daquele da nossa primeira aula, que eu mostrei o funcionamento já interagindo com o banco de dados.

[10:31] Nós vamos ver que, por enquanto, ele está mostrando que o produto é *null*, produto de teste, porque realmente eu ainda não estou conversando com o banco de dados. Então aqui eu só estou simulando. Se eu botar aqui um "aa" em nome do produto e em descrição do produto, e mando salvar, ele me dá uma mensagem de salvo com sucesso, mas nós vamos ver que ele só mostra no console, não vai para o banco de dados.

[10:59] O nosso objetivo agora é qual? É exatamente isso: chegar no `CategoraController` e no `ProdutoController` e desenvolver as regras que vão fazer com que consigamos fazer essa interação com o banco de dados. Mas nessa aula já falamos muito, nós já fizemos um Overview de todo o projeto, de como ele está por enquanto. Esse desenvolvimento vai ficar para o próximo vídeo. Eu vou ficando por aqui. Até lá. Valeu.

Transcrição

[00:00] Olá, aluno. Tudo bom? Agora que nós entendemos o nosso cenário de como a nossa aplicação *view* vai se comunicar com as nossas classes de negócio, que é a nossa Controller, a nossa DAO, chegou a hora de colocarmos a mão na massa e terminar o desenvolvimento das nossas classes que não estão funcionais, que na nossa situação é a `CategoriaController` e a `ProdutoController`.

[00:24] Antes, eu gostaria de lembrar vocês para tomarmos cuidado de não trabalhar no projeto que estávamos utilizando no curso, que é o "loja-virtual-repository", para não correr o risco de fazermos alterações nele achando que estamos fazendo alterações no "loja-virtual-view-repository". Recomendo novamente que fechemos esse projeto "loja-virtual-repository".

[00:42] Para quem não sabe como fechar, é só clicar com o botão direito do mouse em cima do projeto, tem uma opção "Close Project". Fechando o projeto, ele continua no nosso Workspace, mas não conseguimos acessar o projeto. Voltando agora para o nosso "loja-virtual-view-repository", nós vamos começar a mexer no `CategoriaController`.

[01:07] O `CategoriaController`, ele só vai ter um método de `listar()`, que é o responsável por popular aquela nossa `ComboBox` com as categorias, para podermos incluir um novo produto. Essa lista, ela vai ter que chamar de alguma forma a nossa DAO e aí sim fazer a comunicação para devolver a lista de categorias para a nossa *view*. Só que, se vocês lembram, as nossas DAOs, elas esperam uma Connection no momento que instancia o objeto.

[01:44] Então o responsável por fornecer essa Connection vai ser a nossa `CategoriaController`. Para isso, nós vamos usar o construtor. Então, no momento em que a nossa *view* chamar o `CategoriaController`, instanciar o `CategoriaContronller`, a

`CategoriaController` já recupera uma conexão, já instancia a nossa DAO podemos usar essa categoria para chamar as nossas DAO já fazendo essa conexão. Como podemos fazer isso?

[02:11] Vamos lá, vamos criar em `CategoriaController`, um `private CategoriaDAO`, porque nós vamos precisar utilizar essa `CategoriaDAO` no método, então ela vira um atributo da nossa classe. Aqui criamos o nosso construtor, o `public CategoriaController()`. Vamos chamar o `new ConnectionFactory(). recuperarConexao();`, que é o responsável por recuperar a nossa conexão com o seu método `recuperarConexao()`.

[02:32] E ele vai me fornecer uma `Connection connection = new ConnectionFactory(). recuperarConexao();`, que vai ser o que nós vamos fornecer para a nossa DAO. Recuperamos a conexão, chamo aqui agora o `CategoriaDAO`, deixa eu instanciar ele aqui, `new CategoriaDAO(connection);`, ele vai esperar uma Connection, que é a que recuperamos no nosso construtor.

[02:57] E atribuímos essa linha para o nosso `this.categoriaDAO = new CategoriaDAO(connection);`, da nossa classe. Dessa forma já fizemos o necessário para instanciar a nossa DAO passando a conexão, agora o que eu preciso fazer é: vamos chamar a DAO e o método de listar dessa DAO. Então eu vou fazer aqui `public List<Categoria> listar()`, `return this.categoriaDAO.listar();`. Está pronta a nossa classe.

[03:30] Só que eu gostaria de chamar a atenção de vocês aqui para esses warnings que está dando na nossa classe. O nosso método está correto, o nosso construtor está correto, mas ele continua dando esse alerta. É porque, quando criamos as nossas DAOs, tanto no `ConnectionFactory` quanto no `listar()`, nós não fizemos o tratamento da SQL Exception, nós apenas subimos a SQL Exception.

[03:59] Quando fazemos isso, que nós subimos essa exceção, que é quando usamos o `throws SQLException` ou qualquer outra exceção, eu estou avisando para quem me chamar que esse código pode dar um SQL Exception, mas eu não estou tratando. Então o que vai acontecer? Se eu não tratar esse SQL Exception na minha `CategoriaController`, eu vou ter que subir essa exceção para a nossa `ProdutoCategoriaFrame`.

[04:28] Não faz sentido a nossa *view* informar que aquela método, aquela chamada, pode ocasionar uma SQL Exception. O SQL Exception, ele pode dar no momento em que eu faço a operação com o banco de dados, no momento em que eu recupero uma conexão com o banco de dados. Então nada mais justo do que trazermos o tratamento dessas exceções onde realmente pode dar exceção.

[04:54] No nosso caso é a `ConnectionFactory` e a `CategoriaDAO`. Então o que nós vamos fazer aqui? Eu vou tirar esse `throws SQLException`. E nós vamos fazer o seguinte, em `Connection recuperarConexao()`, eu vou fazer um `try`, então tente recuperar uma conexão com o meu banco de dados.

[05:12] Caso você não consiga, podemos dar uma `catch(SQLException e)` e usar um *pattern*, que é para lançar invés de uma exceção checada, nós vamos relançar uma exceção *unchecked*, que uma `throw new RuntimeException()` - calma que o meu computador deu uma travada aqui. Então nós vamos relançar a `RuntimeException(e)` passando o motivo, que pode ser uma SQL Exception.

[05:49] Feito isso nós tratamos o `recuperarConexao()` e nós vemos que sumiu no nosso construtor o problema que nós tínhamos no momento de recuperar a conexão. Vamos fazer a mesma coisa no `listar();` do `CategoriaDAO`. Nós vamos tirar o `throws SQLException` e vamos colocar o nosso `try` em `listar()`.

[06:13] Tentei então recuperar uma lista de categorias, se você não conseguir, vai dar uma SQL Exception e nós vamos relançar com o *pattern*, que é relançando uma exceção *unchecked*. Então uma `catch (SQLException e)`, `throw new RuntimeException(e);`. Está aí o nosso tratamento. Agora, se nós formos ver, o nosso método também parou de reclamar.

[06:48] Uma vez aqui com a `CategoriaController` pronta, nossa `ConnectionFactory` agora com o tratamento de exceção correto, `CategoriaDAO`, o método `listar()` também. Vamos chamar aqui o `ProdutoCategoriaFrame` para ver se vai ter algum problema. Não, desculpa, é no `TestaOperacaoComView`. Vamos chamar ele e vamos ver se ele já está carregando a nossa combo com as categorias.

[07:15] Se formos ver as categorias, eletrônicos, eletrodomésticos e móveis, são as três categorias que nós temos no nosso banco de dados. Funcionou então a ideia de criarmos os nossos Controllers e fazer o tratamento melhor das nossas exceções. Só que agora eu vou deixar o desafio para vocês, para vocês fazerem isso também nos outros métodos da `CategoriaDAO` e em `ProdutoDAO` também.

[07:38] Também vou pedir para vocês, vou desafiar vocês a também criarem a nossa `ProdutoController` com todas essas melhorias que nós fizemos no código. Então na próxima aula nós vamos ver como vai ficar a nossa `ProdutoController` e vamos ver o funcionamento total do sistema, agora com a nossa *view*, Controller e DAO todas implementadas. Então encontro vocês no próximo vídeo.

Transcrição

[00:00] Olá, aluno. Tudo bom? Anteriormente eu deixei um desafio para vocês, para que vocês refatorassem o código com a melhor prática de tratamento de exceção, tratando na DAO, onde realmente a exceção, ela pode ocorrer. Também deixei o desafio para vocês implementarem a `ProdutoController`.

[00:27] Agora nós vamos passar pelo meu código para vocês verem como eu fiz, para vocês verem se está de acordo com o de vocês e executar o projeto, agora com todas as funcionalidades ocorrendo com sucesso. Então primeiro vamos passar pela `CategoriaDAO`, eu fiz no `listarComProduto()` também com o try catch, relançando uma `RuntimeExcepiton`, que é uma exceção *unchecked*.

[00:58] Se você está tendo alguma dúvida referente ao que eu estou falando aqui de *checked* e *unchecked exception*, eu recomendo fortemente fazer o nosso curso de Exceções, que ele é bem completo e vai te dar uma base muito boa para você saber como tratar da melhor maneira essas exceções.

[01:17] Então, na `listarComProduto()` fizemos a mesma coisa que nós tínhamos feito no método `listar()`. Na `ProdutoDAO` fizemos também um tratamento semelhante, com try e catch em cada um dos métodos, então `salvar`, `salvarComCategoria`, todos eles agora são tratados caso ocorra uma SQL Exception no momento de executar a operação, nós já fazemos o tratamento no local, onde é para ser feito.

[01:46] No `ProdutoController`, nós fizemos basicamente aquele mesmo trabalho que nós tínhamos feito com `CategoriaController`, só que usando o `ProdutoDAO`. Então nós instanciamos o `ProdutoDAO`, passando uma Connection que

nós tínhamos recuperado com `recuperarConexao()` e chamamos o método da DAO, o método correspondente de cada DAO na nossa Controller.

[02:15] Então `deletar()` chama o `.deletar()`, `salvar()` chama o `.salvar()`, bem parecido com o que nós tínhamos feito no `CategoriaController`. Feito isso, é para o nosso projeto estar 100% funcional, agora nós vamos poder ver isso no `TestaOperacaoComView`, mandando executar o projeto e vamos dar uma navegada na nossa tela para ver se está tudo funcionando corretamente.

[02:40] Subimos a nossa aplicação. Então agora nós já vemos que nós estamos listando os produtos da nossa base de dados. Vamos ver os comandos, se estão todos funcionais, então se eu mandar limpar aqui, ele tem que zerar as caixas de texto. Aparentemente ok. Vamos inserir um novo produto, vamos colocar o "Microfone", "Microfone para computador", a categoria dele vai ser um eletrônico.

[03:08] Se eu mandar salvar, salvo com sucesso. O microfone está inserido na *grid* e, consequentemente, no nosso banco de dados. Agora o nosso "Celular com câmera" acabou no estoque, vamos editar para mostrar um "Celular sem câmera", para mostrar que é o que nós temos disponível. Mando alterar, fez a alteração, certo.

[03:30] E o "Sofá" não é um item que eu vou vender mais na minha loja, eu quero excluir esse item para não mostrar mais para os clientes. Mando excluir e está aí: item excluído com sucesso. Apenas três produtos agora no nosso estoque. Bom, pessoal, então com isso nós terminamos o desenvolvimento da nossa tela, nós conseguimos entender como que funcionam essas camadas, tudo certo.

[03:53] O porquê temos uma Controller, que é exatamente para tirar a responsabilidade que não é da *view* da *view*, nós passamos para essa Controller, então ela que vai abrir conexão, ela que vai comunicar com a DAO e ela só retorna para a *view* as informações necessárias para que a *view* mostre para a tela, que é o objetivo dela.

[04:16] As nossas classes, elas tem que ter responsabilidade única, então a *view* só mostra informação, a Controller faz esse controle ali no meio, então pega a requisição, manda para a DAO correta. A DAO sim vai no nosso banco de dados, que ela é

responsável por isso. Então nós vimos que o nosso projeto fica muito bem desenvolvido.

[04:36] Então espero que vocês tenham gostado desse incremento, dessa nossa tela. Qualquer dúvida que vocês tiverem sobre alguma coisa que nós fizemos aqui na aula, mande no nosso fórum que nós estaremos à disposição para retirá-las. E, no mais, é isso. Vejo vocês no próximo vídeo. Valeu.

09

O que aprendemos?

Nessa aula aprendemos:

- uma aplicação é escrita em camadas
  - camadas clássicas são *view*, *controller*, *modelo* e *persistência*
- o fluxo entre as camadas segue a ordem:

```
view <--> controller <--> persistencia
```

COPIAR CÓDIGO

- nesse curso focamos na camada de persistência
- uma camada não deve deixar "vazar" detalhes da implementação (por exemplo uma exceção como `SQLException`)
- em outras formações você aprenderá como criar a **view** ou **front-end** para Android (*mobile*) ou web (*html*)

Transcrição

[00:00] Olá, aluno. Tudo bom? Para você que chegou até aqui, meus parabéns. Nós tivemos momentos de muito aprendizado. Nós vimos como fazer a nossa aplicação interagir com o banco de dados, nós aprendemos sobre padrões para que essa comunicação seja feita da melhor maneira possível, com o código legível, com o código de fácil manutenção.

[00:23] Então nas nossas DAOS, nós conseguimos concentrar os métodos que irão fazer aquelas operações com o banco de dados, como o select, o insert, o update. Além disso, nós vimos como nós podemos não passar por problemas, igual o query N+1, e apenas usando uma melhora na nossa query, que são os joins, no nosso caso, nós fizemos o Inner join com a tabela Produtos e Categoria.

[00:54] Vimos que em vez de eu ir no banco de dados várias vezes, fazendo com que eu possa ter uma queda no meu banco de dados, eu possa ter uma queda de performance nesse banco de dados, eu vá apenas uma vez e traga as informações que eu preciso, e eu posso trabalhar com ela da maneira que eu bem entender.

[01:16] Então após passar por tudo isso, nós vimos como podemos criar uma tela para conseguirmos fazer essa interação com o banco de dados. O nosso usuário, ele não vai botar a mão no código, ele vai botar a mão em uma interface gráfica, seja ela web, que nós vimos que pode ser o nosso próximo passo, seja ela mobile ou até desktop, como nós fizemos, que não é tão recomendado, mas que funciona.

[01:49] E nós vimos que para isso, nós temos que tirar códigos que não são de responsabilidade da nossa *view*, então esses códigos, eles vão ficar em uma camada própria, para que o nosso código fique sempre muito bem organizado. Então, só para

relembrarmos, eu vou novamente executar a nossa tela, para vermos o quão legal ficou podermos fazer as nossas operações a partir de uma interface, para facilitar a vida do usuário.

[02:25] Então, dito isso, mais uma vez eu os parabenizo por ter chegado até aqui. Não é fácil fazer um curso com tantos recursos, muitas vezes pode ser cansativo, mas eu acredito que isso vai ajudar bastante vocês na carreira como desenvolvedor. Nós sempre falamos que se tiver qualquer dúvida em algum ponto do curso, seja nas aulas, seja nos exercícios, podem nos mandar no fórum, vamos estar à disposição para auxiliar vocês.

[03:02] E, como foi informado durante as aulas, não deixem de fazer as próximas formações. Agora você pode querer ser um especialista mobile, nós temos a formação para isso. Você pode querer ser um especialista em desenvolvimento web, também temos uma formação para isso. A ideia é que vocês adquiram cada vez mais conhecimento através da plataforma. Então eu vou ficando por aqui e vejo você no próximo curso.