PÓS-GRADUAÇÃO ALFA

UNIALFA
CENTRO UNIVERSITÁRIO
ALVES FARIA

Diego Américo Guedes

PROGRAMAÇÃO COM FRAMEWORKS E COMPONENTES

GRUPO JOSÉ ALVES

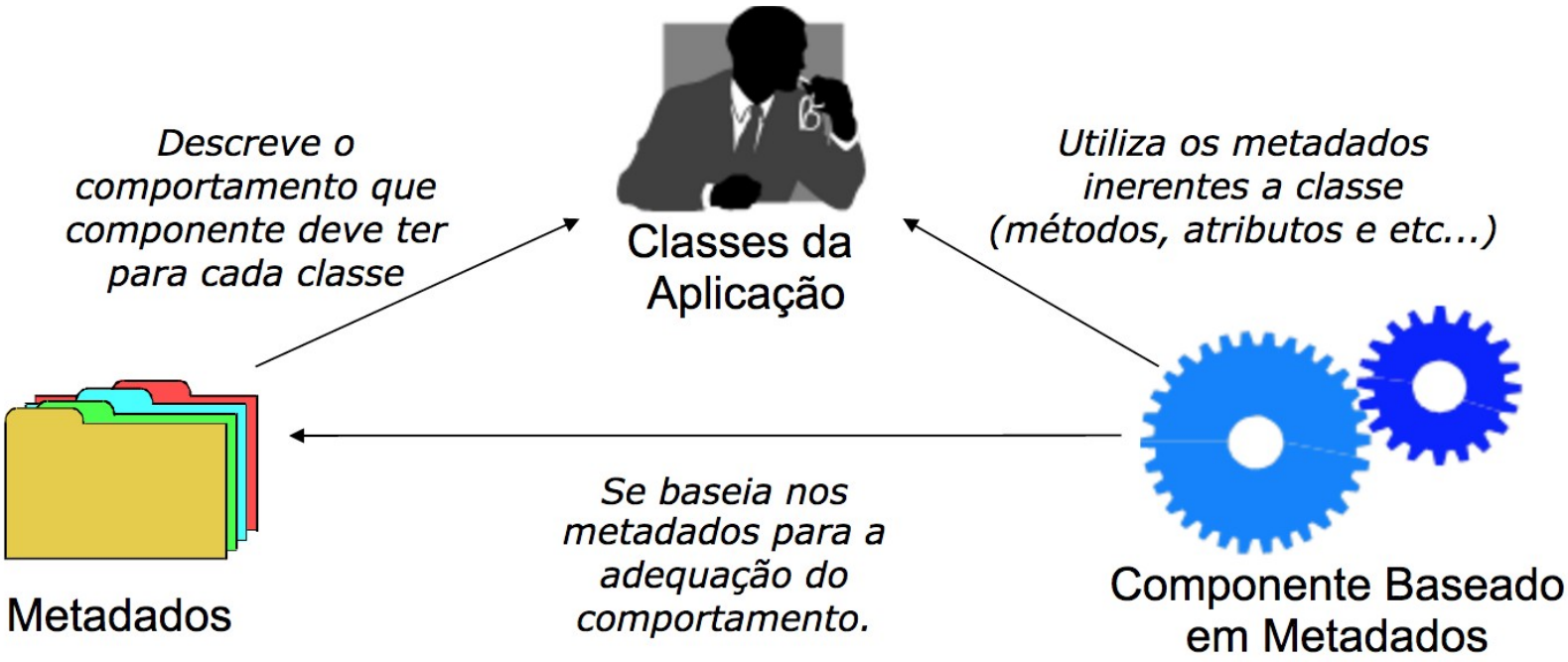
Framework Baseado em Metadados

Prof. Me. Diego Guedes

Definição

- Framework nos quais o seu comportamento é configurado por meio de metadados (dados que definem outros dados)

Uso de metadados



- Os metadados podem ser da aplicação (onde eles configuram características do componentes da aplicação) ou de objetos (onde eles configuram as características de cada objeto da aplicação).

Vantagens

- É possível personalizar o comportamento de um componente ou framework de acordo com as necessidades da aplicação
- Aumento de reuso do componente que utilizar os metadados, devido a possibilidade de ser configurado
- Diminuição de código repetitivo e braçal, o que diminui a possibilidade de defeitos muitas vezes difíceis de serem encontrados
- Aumento da produtividade da equipe
- Aumenta a flexibilidade da aplicação devido ao caráter configurável do componente
- Dependendo da forma que os metadados forem armazenados, eles podem ser alterados sem a necessidade de recompilar o código fonte

Exemplos

- **Hibernate**: uso de metadados para mapear o paradigma orientado a objetos para o paradigma relacional, provendo um componente genérico para o acesso a dados
- **Spring**: um framework baseado em metadados para o gerenciamento da injeção de dependência e inversão de controle em uma aplicação
- **EJB 3**: Uso de metadados para gerenciamento do ciclo de vida, controle de transações, segurança, injeção de dependência e outras coisas.

Já trabalhos com Spring.
Agora, vamos trabalhar com o Hibernate ! \o/

Hibernate

Motivação

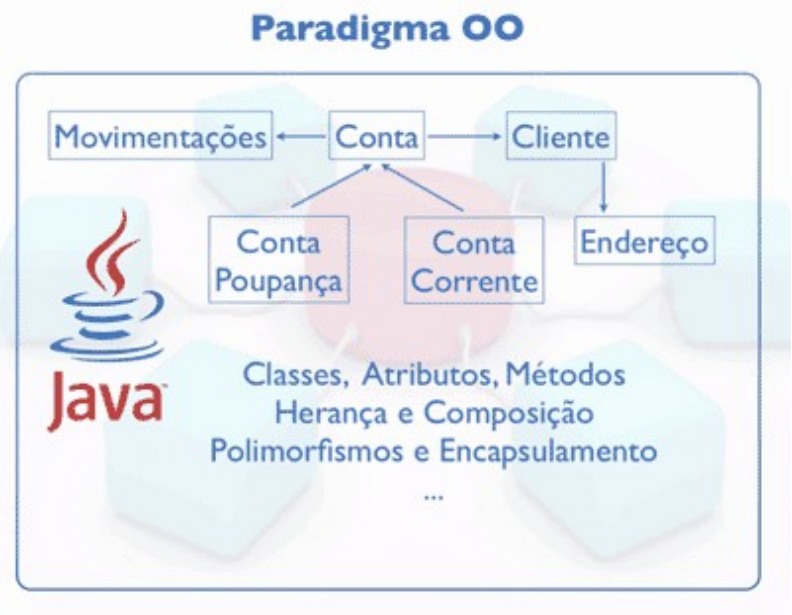
- Atualmente grande parte das aplicações devem integrar-se com um banco de dados.
- Ao usar um banco geralmente trabalhamos com o **paradigma relacional**, que nos provê mecanismos para relacionar as diferentes informações que o nosso sistema armazenará
- Para representarmos as informações no banco, utilizamos **tabelas** e **colunas**.
- As tabelas geralmente possuem **chaves primárias** (PK) e podem ser relacionadas por meio da criação de **chaves estrangeiras** (FK) em outras tabelas

Motivação

- Conseguimos trabalhar com o banco alterando e consultando informações através do SQL (*Structured Query Language*), que, apesar de ser um padrão, apresenta diferenças significativas dependendo do fabricante

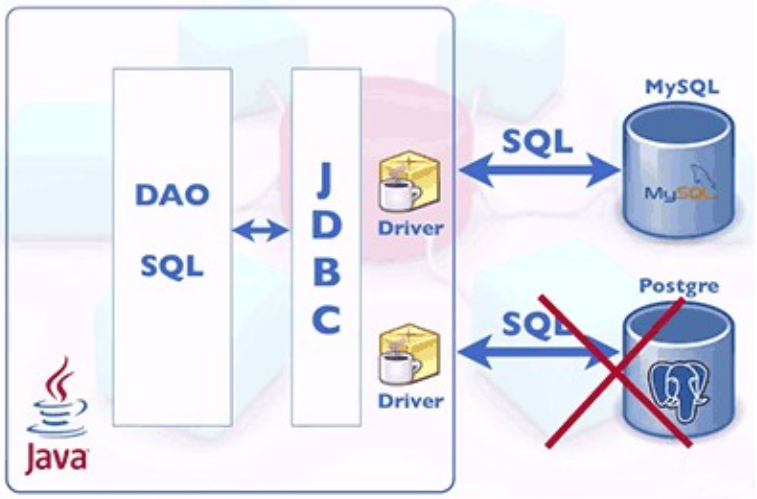
O paradigma orientado a objetos

- Quando trabalhamos com uma aplicação Java, seguimos o **paradigma orientado a objetos**, onde representamos nossas informações por meio de **classes**, **atributos** e **métodos**.
- Além disso, podemos utilizar **herança** e **composição** para se relacionar, encapsular dados e lógicas, aproveitar o polimorfismo, entre outras possibilidades.



Evitando o SQL dentro do código Java

- **API JDBC** nos provê formas de executarmos comandos SQL dentro do código Java
- O problema do JDBC é que precisamos escrever código **SQL misturado com o nosso código Java**
  - Sempre que quisermos mudar qualquer SQL da nossa aplicação, precisamos alterar a classe e recompilá-la
  - o SQL nem sempre é portável entre bancos de dados diferentes e ao escrevê-lo podemos ficar presos naquele fornecedor



JDBC e SQL - problemas de manutenção

- Vamos provar isso através do projeto **contas-jdbc**
- Na pasta lib está o driver de conexão do **MySQL** e o JAR do banco de dados **HSQLDB**
- Queremos executar as operações básicas no banco de dados, ou seja, inserir, alterar e listar contas
- Ao abrir a classe **TesteJDBC** podemos ver como adicionar uma conta.
- Após sua criação, usamos o DAO para executar o SQL.
- O código é simples, mas vamos também analisar o DAO.
- Qualquer alteração no modelo terá um grande impacto nessa classe. Isso pode ser ilustrado comentando o atributo *titular* e seus *getter* e *setter*. Repare que praticamente todos os métodos do ContaDAO param de compilar.

Persistir objetos sem escrever SQL

- O ideal seria que trabalhássemos apenas com a nossa linguagem de programação e todo o trabalho do banco de dados que envolve SQL, transação e outros cuidados fossem abstraídos.
- Ou seja, para "gravarmos" a conta no banco queremos utilizar o mínimo de código possível, algo como:
  - conexao.salva(conta);
- **Objetivo**: diminuir o uso de detalhes do **paradigma relacional** para programar mais focado com o **paradigma orientado a objetos**.

Unindo os dois mundos através do Mapeamento Objeto-Relacional

- O problema é que o paradigma relacional é bem diferente do orientado a objetos.
- Essa diferença, também conhecida como *impedância* (*impedance mismatch*), é abstraída pelos frameworks que mapeiam o mundo OO automaticamente para mundo relacional - são os famosos *frameworks ORM (Object Relational Mapping)*

A Java Persistence API - JPA

- Com o passar do tempo, surgiram diversos frameworks ORM, cada um funcionando de maneira diferente, mas o que se destacou no mercado foi o **Hibernate**, sendo fácil de usar sem abrir a mão do poder dos bancos de dados relacionais.
- O **Hibernate** tornou-se praticamente o padrão do mercado, levando a criação da especificação **JPA** (*Java Persistence API*) dentro do JavaEE

O primeiro exemplo com JPA

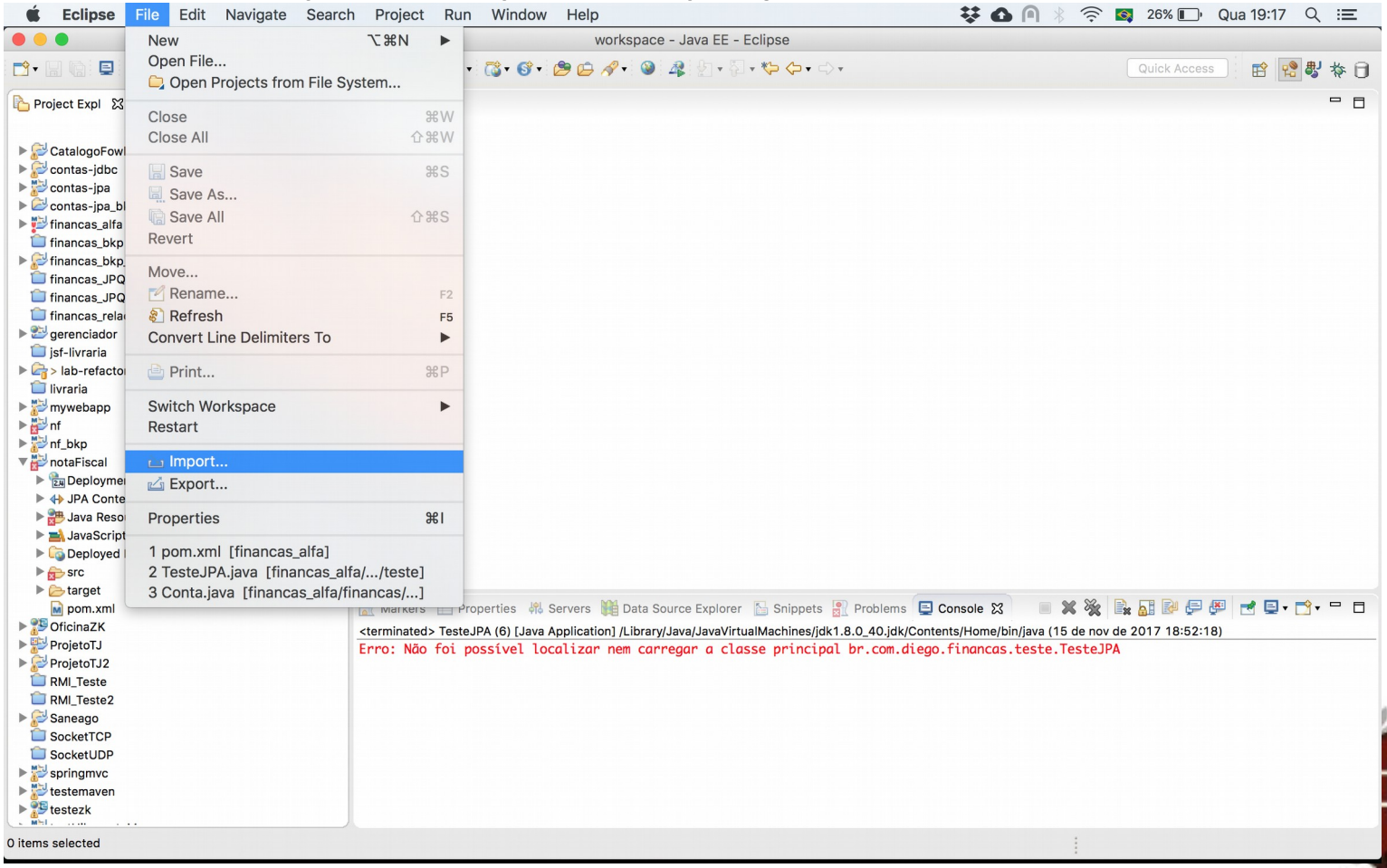
- Vamos mostrar os benefícios do JPA com Hibernate através de um pequeno projeto (**contas-jpa**)
- Configuramos os dados da conexão dentro do arquivo **persistence.xml**
- Nele encontram-se as informações típicas sobre o Driver, URL, login e senha utilizados

Configuração e inicialização do JPA

Prof. Me. Diego Guedes

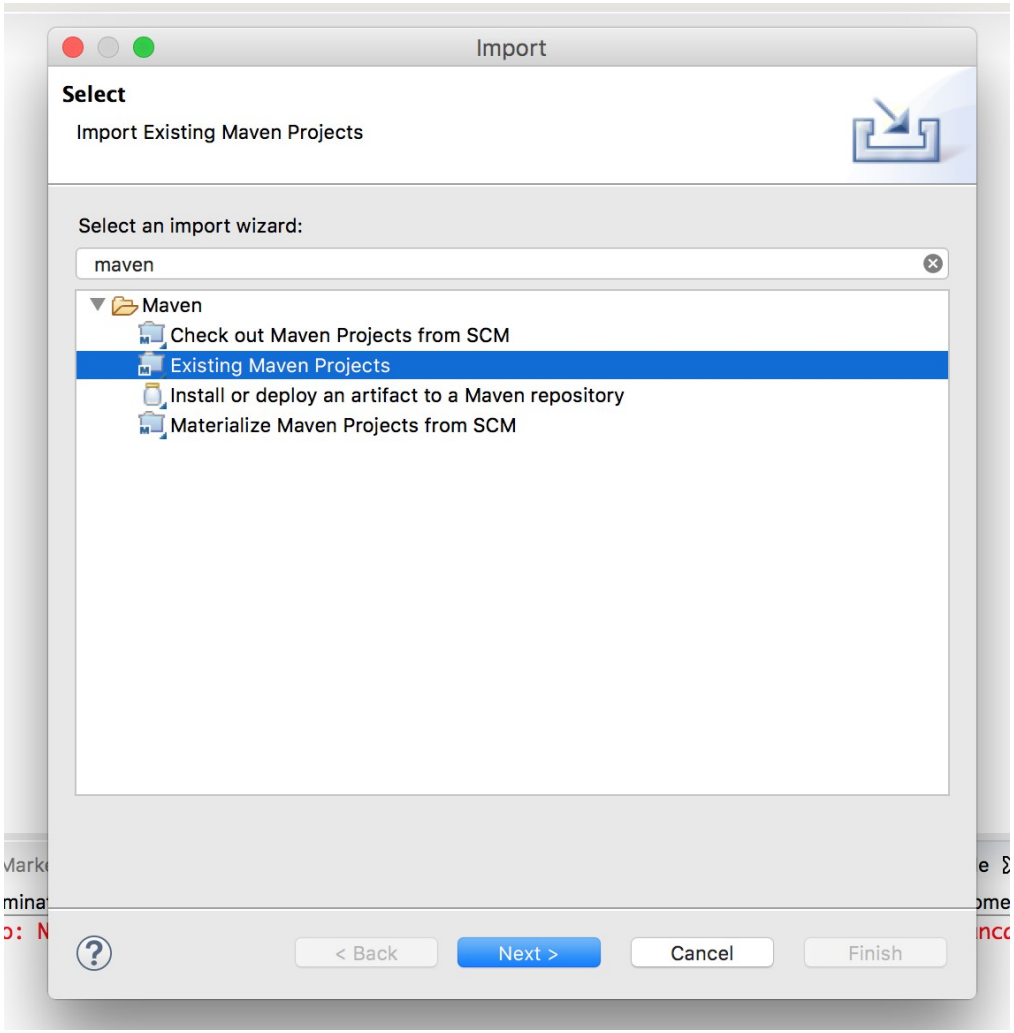
Abrindo o projeto

- Abra o eclipse e importe o projeto



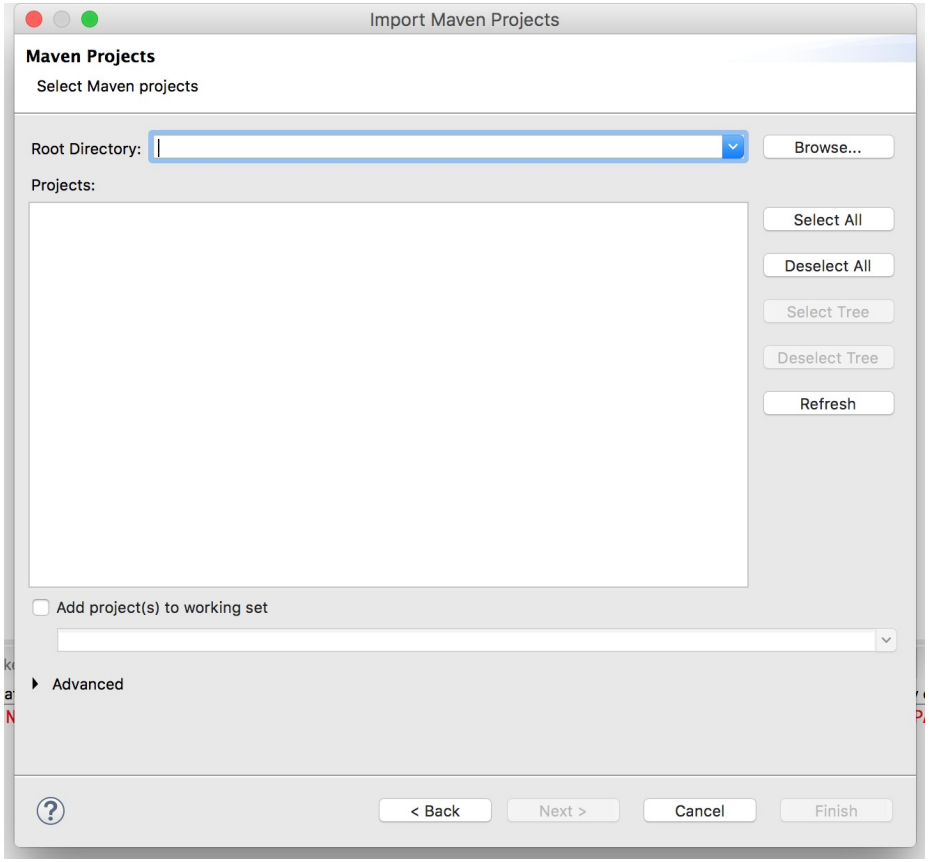
Abrindo o projeto

- Existing Maven Projects



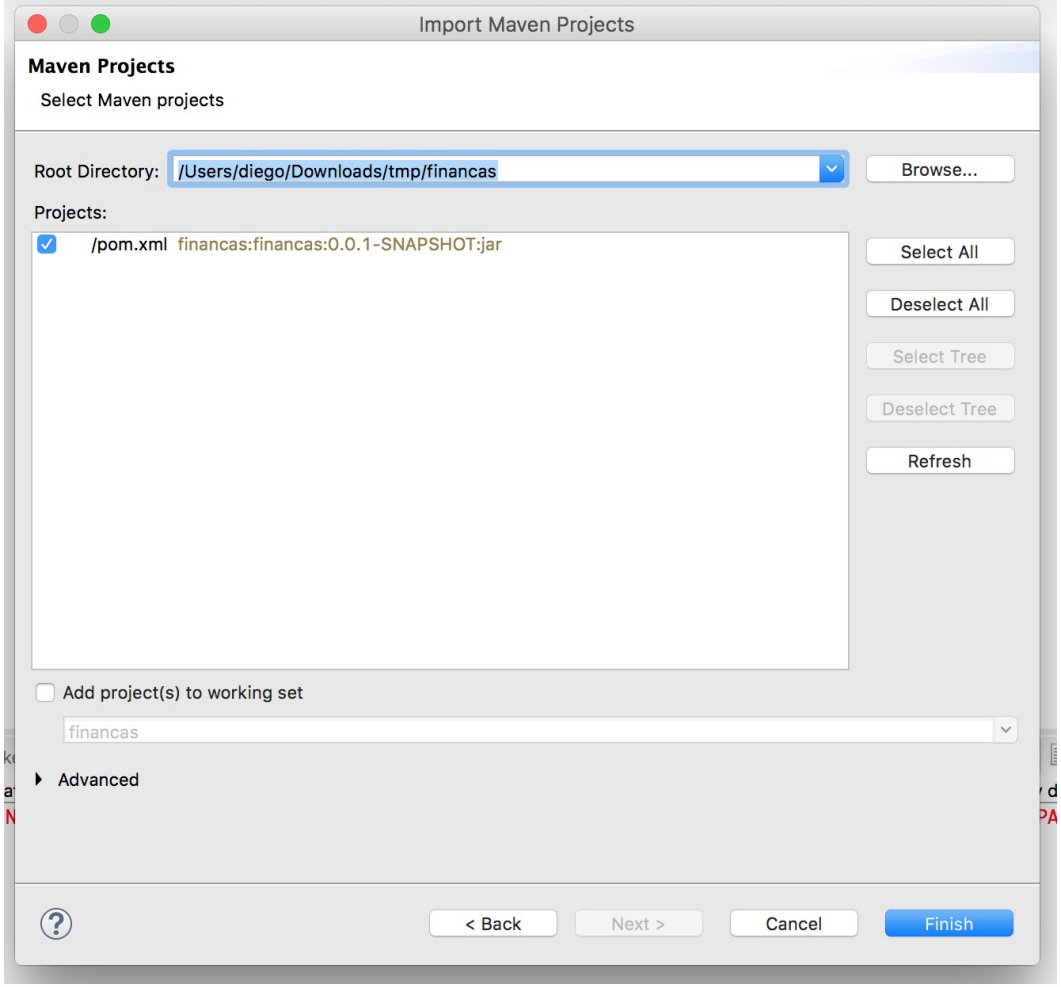
Abrindo o projeto

- Escolha o diretório onde baixou o projeto



Abrindo o projeto

- Por fim, Finish! :-)



Bibliotecas do JPA e Hibernate

- Como é um projeto Maven, todas as dependência que precisaremos irão ser baixadas e adicionadas no projeto

Preparação do modelo

- Preparando o modelo
  - Na pasta src criaremos uma nova classe Conta dentro do pacote br.com.diego.financas.modelo
  - É essa classe que queremos mapear para uma tabela
  - Vamos criar 5 atributos: id, titular, numero, banco e agencia, todos do tipo String, exceto id que é Integer.

A primeira Entidade

- Como usaremos Postgres deixamos a estratégia como sequence
  - @GeneratedValue(strategy = GenerationType.AUTO, generator = "SEQ_CONTAS")
- Com outros bancos, por exemplo, Oracle poderíamos usar sequências e mysql identity

A primeira Entidade

- Para mapear a classe Conta é preciso anotá-la com @Entity do pacote javax.persistence
- Entidades (ou *Entities*) são as classes que tem uma tabela associada
- Além disso, é obrigatório declarar o atributo que representa a chave primária com @Id
- A anotação @GeneratedValue é opcional, mas é muito comum usar com chaves auxiliares.
- Como iremos usar sequence, devemos definí-la
  - @SequenceGenerator(name = "SEQ_CONTAS", sequenceName = "SEQ_CONTAS", initialValue = 1)
- Com ela indicamos que o banco deve atribuir o valor da chave, e não a aplicação.
- Ao inserir uma conta no banco, automaticamente será alocada uma *ID*

Configuração no persistence.xml

- Para definir os dados de conexão, o JPA possui um arquivo de configuração, o persistence.xml.
- Pela especificação esse arquivo deve ficar dentro da pasta META-INF.
- Vamos criar essa pasta (META-INF) dentro do src copiando para dentro dela o arquivo persistence.xml que você já baixou do github
- No persistence.xml toda configuração fica dentro de um elemento persistence-unit.
- Uma unidade de persistência possui um nome e dentro do mesmo persistence.xml poderiam ter várias unidades, por exemplo, para bancos de dados diferentes.

Preparação do banco e a geração do esquema

- Antes de testar o JPA, não pode-se esquecer de criar o banco no Postgres.
- **O Hibernate gera as tabelas e colunas, mas não o banco em si.**

Inicialização do JPA e Hibernate

- criaremos uma classe auxiliar chamada **TesteJPA**, no pacote br.com.diego.financas.teste, com um simples método main
- A primeira coisa a fazer é carregar a configuração, aquele arquivo persistence.xml.
- O JPA possui uma classe com o mesmo nome: Persistence.
- Usaremos ela para criar uma EntityManagerFactory baseada na unidade de persistencia financas, chamando o método createEntityManagerFactory(...) da classe Persistence.
  - O parâmetro do método createEntityManagerFactory é o nome da unidade de persistencia que, no nosso caso, é contas-postgres
- A fábrica por sua vez cria um EntityManager, por meio do método createEntityManager().
- Também já vamos fechar o EntityManager através do método close().

Inicialização do JPA e Hibernate

- O EntityManager possui os principais métodos do JPA
- Através do EntityManager podemos, por exemplo, persistir uma entidade. O método que possui essa responsabilidade chama-se persist().
- Para o código compilar falta instanciar um objeto da classe Conta. Vamos preencher os atributos titular, banco, numero e agencia

Gerenciamento de estados pelo EntityManager

Prof. Me. Diego Guedes

Gerenciamento da Transação

- É preciso cuidar da transação.
- O EntityManager possui o método getTransaction(), que devolve um objeto que representa a transação.
- Com ele em mãos vamos iniciar (begin), comitar (commit) e fechar (close) a transação.

Apenas uma EntityManagerFactory

- No pacote br.com.diego.financas.util, vamos criar uma classe chamada JPAUtil, que ajuda na inicialização do JPA, garantindo que exista apenas uma EntityManagerFactory
  - Padrão de projeto *Singleton*
- Para facilitar, inserimos algumas contas no banco através da classe **PopulaConta**.

Carregar entidade pela chave primária

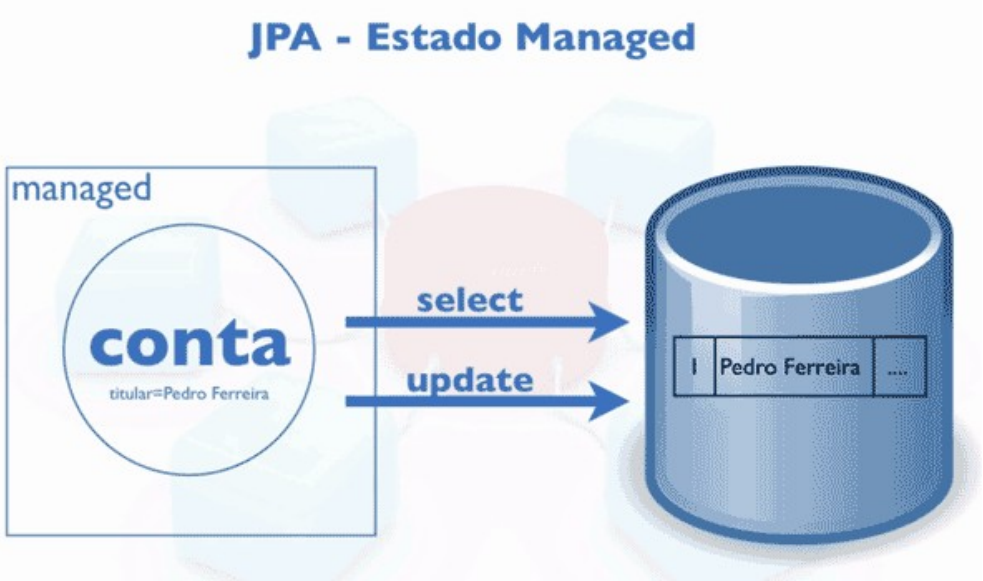
- O EntityManager provê o método find com a finalidade carregar uma entidade por sua chave primária.
- O find recebe dois parâmetros.
  - O primeiro é a classe da entidade que queremos carregar
  - o segundo, o valor da chave primária
- O retorno é um objeto da classe inicializado com os valores do banco de dados
- **Cuidado! O tipo da chave primária deve bater com o tipo definido na classe Conta**

Entidades gerenciadas - O estado Managed

- No exemplo só acessamos a conta.
- Vamos alterar uma vez algum valor. Por exemplo, o titular:
  - conta.setTitular(”Pedro Henrique");
  - System.out.println(conta.getTitular());
- Ao executar o método main percebemos que a **conta não só foi carregada, como também alterada (veja o log)**

Entidades gerenciadas - O estado Managed

- JPA verificou se houve alguma alteração na entidade e executou um update.
- O método find devolveu um conta gerenciada(ou **Managed**) pelo JPA.
- Neste estado é garantido que o objeto terá sua representação idêntica no banco.



Entidades gerenciadas - O estado Managed

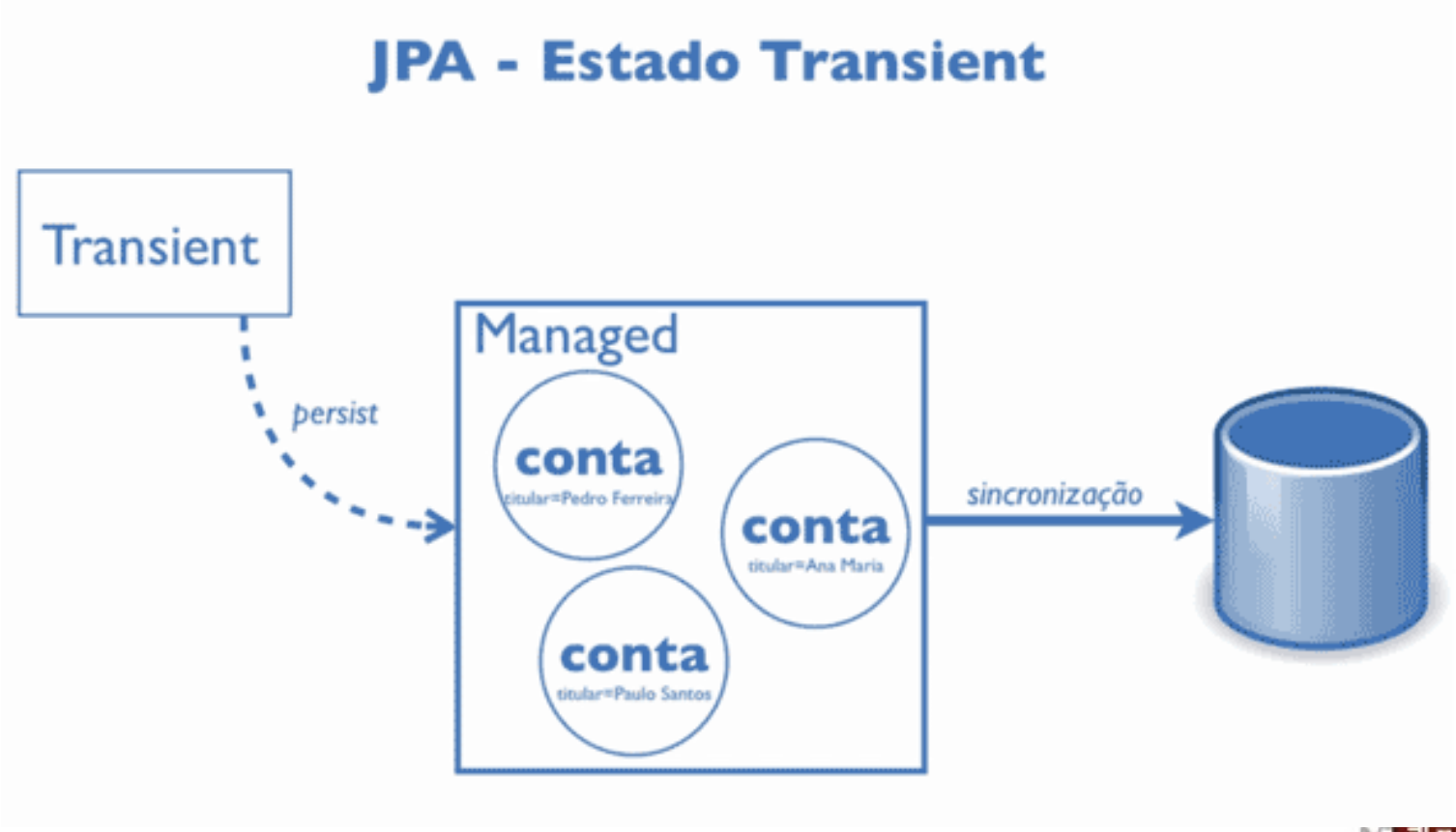
- Repare que ao repetirmos a execução, não será feito nenhum update, pois a conta em memória agora é igual a conta no banco de dados.
- Com o JPA, o objetivo é sempre trazer os objetos para o estado **Managed**, já que assim eles serão gerenciados e automaticamente sincronizados com o banco

Persistindo objetos transientes

Persistindo objetos transientes

- Nem sempre queremos carregar um objeto que já exista no banco de dados
- Repare que a conta não existia no banco.
- Ela foi criada pela aplicação e até que fosse chamado o método persist() do JPA, ela não seria salva no banco e sumiria totalmente se caso a aplicação terminasse.
- Esse estado é chamado **Transiente**(ou ***Transient***) e a tarefa do método persist() é justamente alterar esse estado para **Gerenciado**(***Managed***).

Persistindo objetos transientes

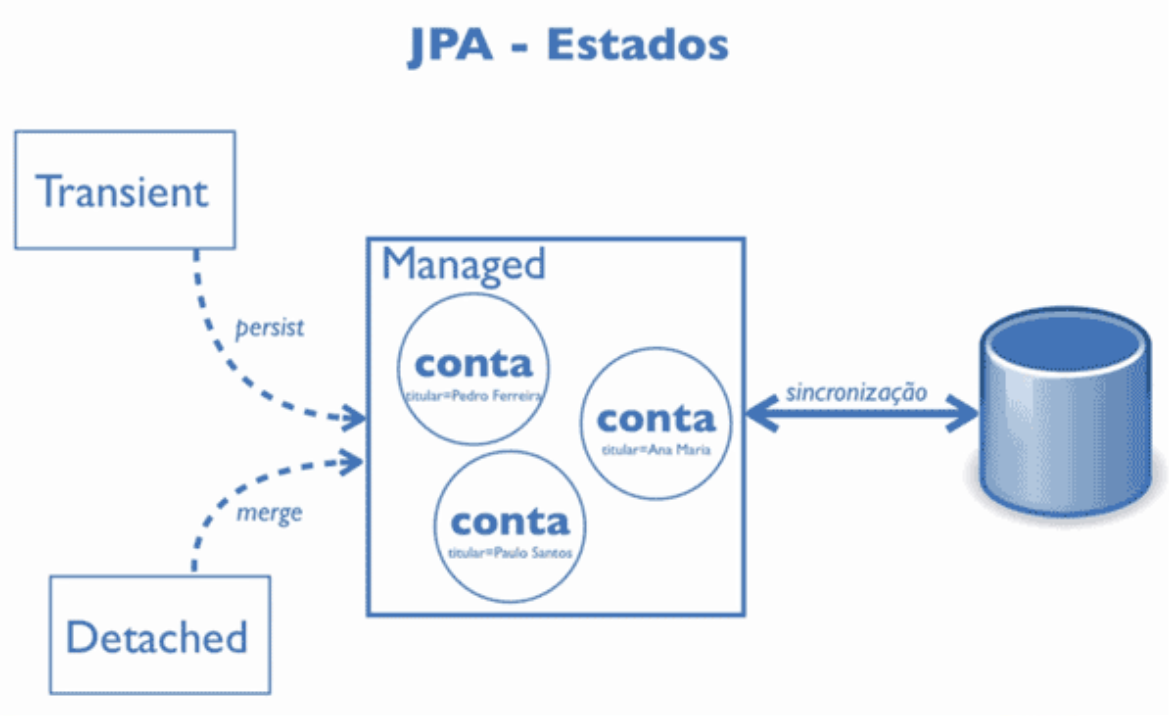


É importante frisar que o estado *Managed* da entidade dura enquanto o EntityManager estiver aberto.

Atualizar objetos desatachados

Objetos desatachados

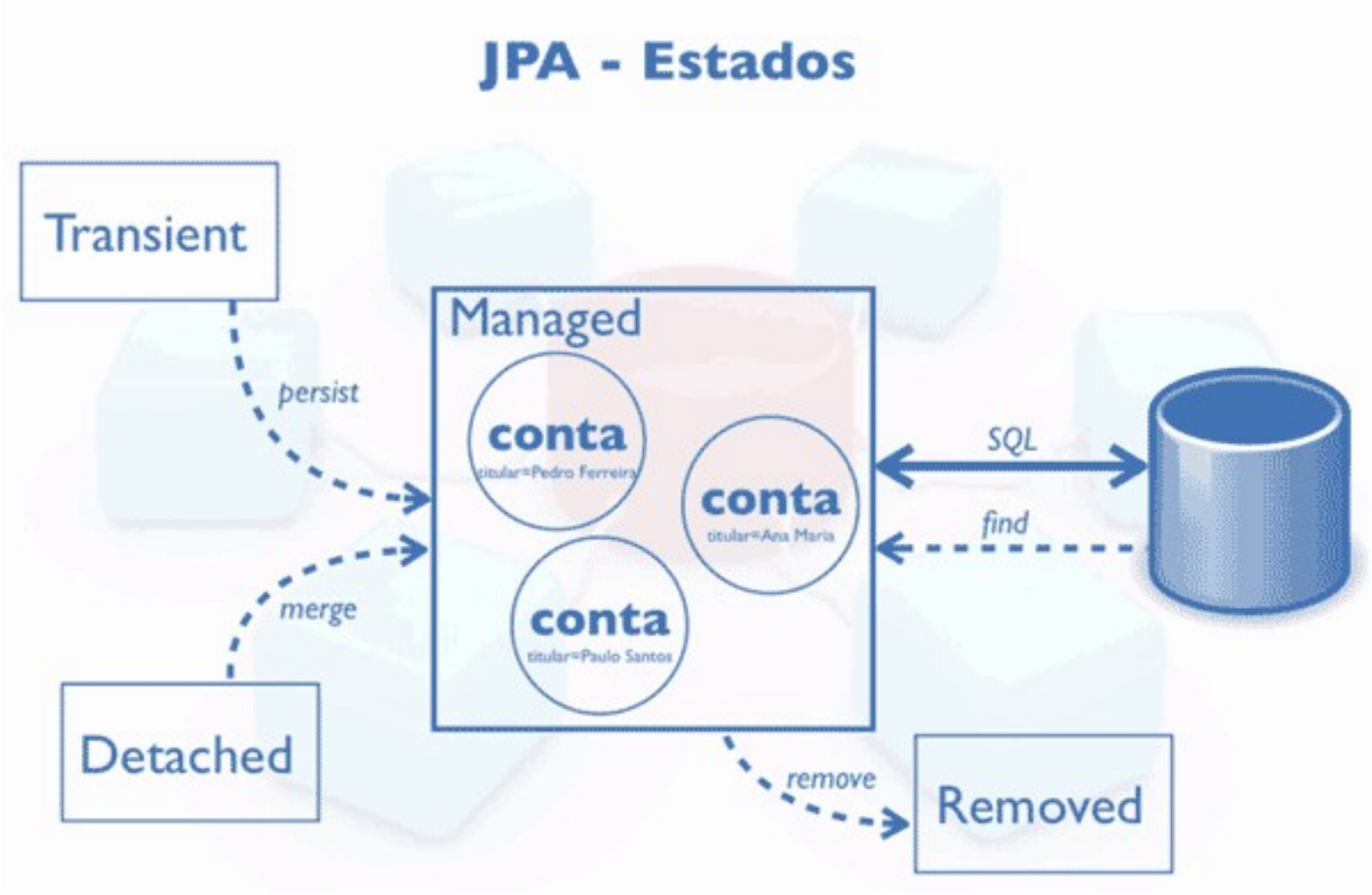
- A entidade representa algo que possivelmente está no banco de dados, mas o *EntityManager* o desconhece: a entidade está fora do contexto, *detached*.
- Como vimos, a tarefa do desenvolvedor é deixar as entidades *Managed*.
- Para fazer com que um objeto *Detached* volte a ser *Managed*, devemos usar o método merge().



Removendo entidades pelo EntityManager

- Por último falta entender como remover uma entidade.
- Para tal, o JPA possui um método remove() que recebe a entidade a ser removida

Resumo dos estados



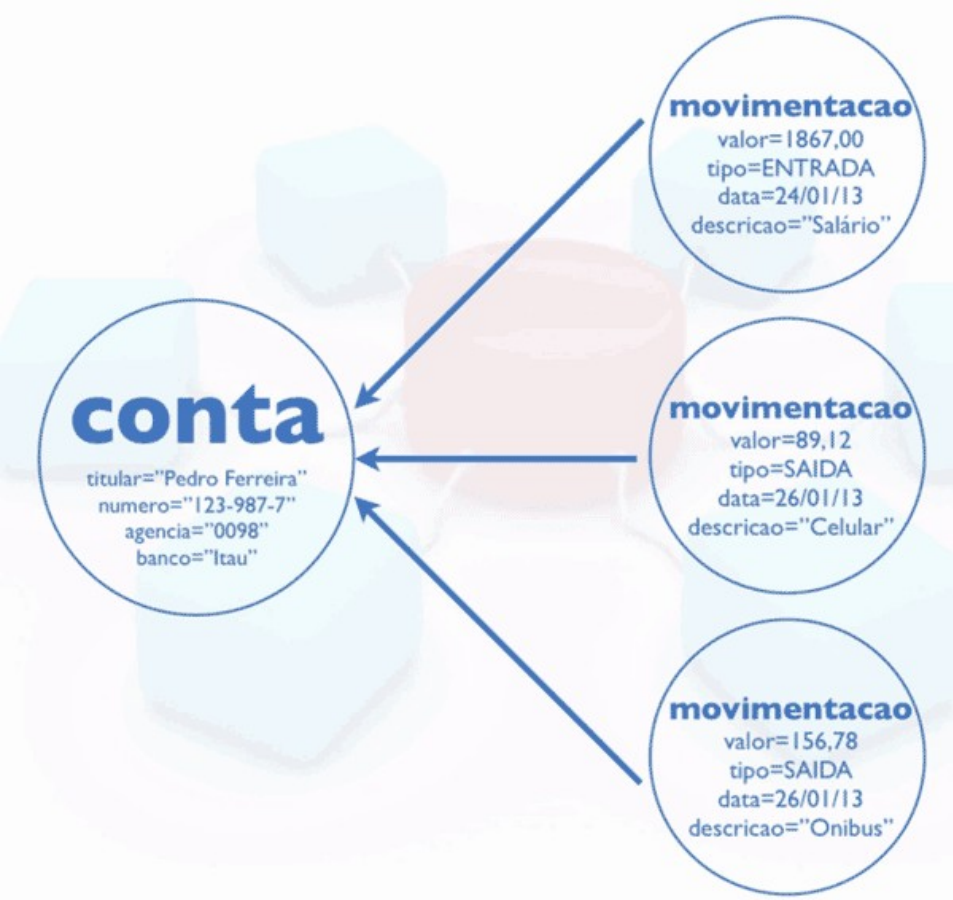
Relacionamento entre Entidades

Prof. Me. Diego Guedes

Relacionamento entre Entidades

- Sempre que trabalhamos com uma conta bancária, realizamos pagamentos, recebemos salários, transferimos dinheiro para outras contas e assim por diante.
- As operações que realizamos em nossas contas são também conhecidas como movimentações, que a partir de agora vamos representar em nosso sistema.
- Cada **Movimentacao** deve ter um valor movimentado, tipo, data de realização, descrição e também uma Conta vinculada.

Relacionamento entre Entidades



Relacionando a Movimentação com uma Conta

- Vamos criar uma nova entidade chamada **Movimentacao** que ficará no pacote br.com.diego.financas.modelo

Mapeando datas e enumerações

- O tipo de movimentação poderia ser uma String, onde guardaríamos ENTRADA ou SAIDA como valor. Qual o problema?

Mapeando datas e enumerações

- O tipo de movimentação poderia ser uma String, onde guardaríamos ENTRADA ou SAIDA como valor. Qual o problema?
  - Uma String pode receber qualquer valor, não só esses dois, assim perderíamos a confiabilidade dos dados
  - Qual uma possível solução?

Mapeando datas e enumerações

- O tipo de movimentação poderia ser uma String, onde guardaríamos ENTRADA ou SAIDA como valor. Qual o problema?
  - Uma String pode receber qualquer valor, não só esses dois, assim perderíamos a confiabilidade dos dados
  - Qual uma possível solução? **Usar Enum!**

Mapeando datas e enumerações

- O tipo de movimentação poderia ser uma String, onde guardaríamos ENTRADA ou SAIDA como valor. Qual o problema?
  - Uma String pode receber qualquer valor, não só esses dois, assim perderíamos a confiabilidade dos dados
  - Qual uma possível solução? **Usar Enum!**
- Como agora o atributo é uma Enum, precisamos anotá-lo com @Enumerated(EnumType.STRING)

Mapeando datas e enumerações

- Quando trabalhamos com datas, normalmente usamos o tipo Calendar.
- Anotar o atributo com **@Temporal**, depois definir o parâmetro de precisão desejado (*TemporalType*). Aqui temos 3 opções:
  - **DATE:** somente a data, sem a hora;
  - **TIME:** somente a hora, sem data;
  - **TIMESTAMP:** tanto data quanto hora

Definindo a cardinalidade do relacionamento

Definindo a cardinalidade do relacionamento

- O JPA sempre tentará mapear os tipos do Java para os tipos adequados no banco de dados.
- Dessa forma, ao criar um atributo do tipo String no Java, ele será mapeado para "*character varying*" no Postgres.
- Isso vale para os principais tipos do Java.

Definindo a cardinalidade do relacionamento

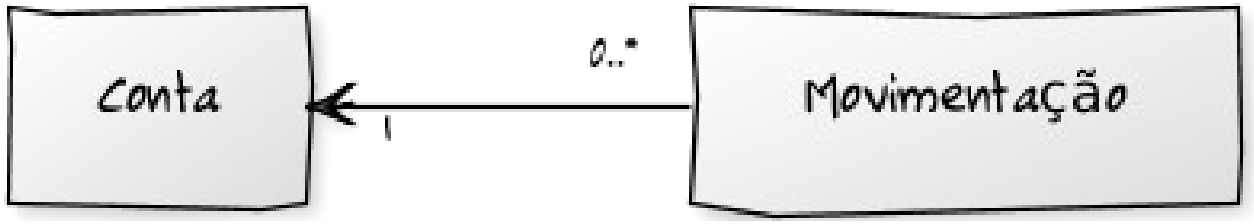
- No entanto, nossa entidade **Movimentacao** terá um atributo Conta.
- Como o JPA mapeará esse atributo no banco de dados? Um varchar ou um int?

Definindo a cardinalidade do relacionamento

- No entanto, nossa entidade **Movimentacao** terá um atributo Conta.
- Como o JPA mapeará esse atributo no banco de dados? Um varchar ou um int?
- O que queremos mapear é um **relacionamento** entre as entidades, que no banco deverá ser refletido por uma chave estrangeira (*Foreign Key*)
- A única informação que precisamos dizer é qual a cardinalidade da Movimentacao em relação à Conta.

Definindo a cardinalidade do relacionamento

- Esse relacionamento é de **muitos para um**.
- Essa cardinalidade nós representamos no código Java com a anotação **@ManyToOne**.
- Vamos então anotar nosso atributo conta:



Definindo a cardinalidade do relacionamento

- Com isso, a implementação do JPA gerará uma nova coluna na tabela Movimentacao que se chamará conta_id.

- **Não podemos esquecer de registrar a nova entidade Movimentacao no arquivo persistence.xml. Vamos editar o arquivo persistence.xml e acrescentar na *persistence-unit* a nova entidade:**

Lidando com a TransientPropertyValueException

- Para nossa supresa, será lançado uma **IllegalStateException** que conterá como causa uma **TransientPropertyValueException**. O que estará errado com nosso código?

Lidando com a TransientPropertyValueException

- Para nossa supresa, será lançado uma **IllegalStateException** que conterá como causa uma **TransientPropertyValueException**. O que estará errado com nosso código?
  - Essa exceção indica que a nossa conta ainda está transiente e, por isso, o relacionamento não pode ser fechado no banco de dados.
  - Qual a solução?

Lidando com a TransientPropertyValueException

- Para nossa supresa, será lançado uma **IllegalStateException** que conterá como causa uma **TransientPropertyValueException**. O que estará errado com nosso código?
  - Essa exceção indica que a nossa conta ainda está transiente e, por isso, o relacionamento não pode ser fechado no banco de dados.
  - Qual a solução? **persistir a Conta antes de persistir a Movimentacao**

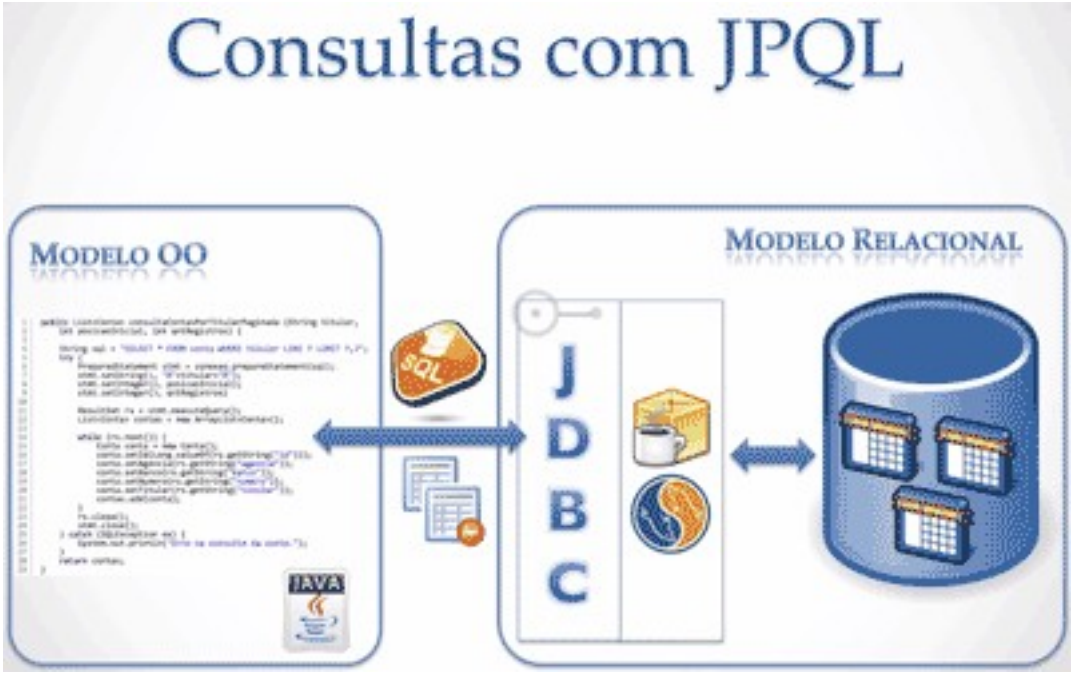
Consultas com o Java Persistence Query Language

Prof. Me. Diego Guedes

Consultas com o Java Persistence Query Language

- Por padrão, todo banco de dados relacional aceita a *Structured Query Language (SQL)* para realizar consultas e manipulações em modelos relacionais
- Quando adotamos o JPA, uma das coisas que buscamos é nos distanciar o máximo possível do modelo relacional e focar exclusivamente no modelo orientado a objetos e seus estados

- Além disso, o **SQL** nos obriga a conhecer bem o modelo de dados, e podemos facilmente notar isso analisando um simples código de consulta que usávamos quando trabalhávamos diretamente com a API do JDBC

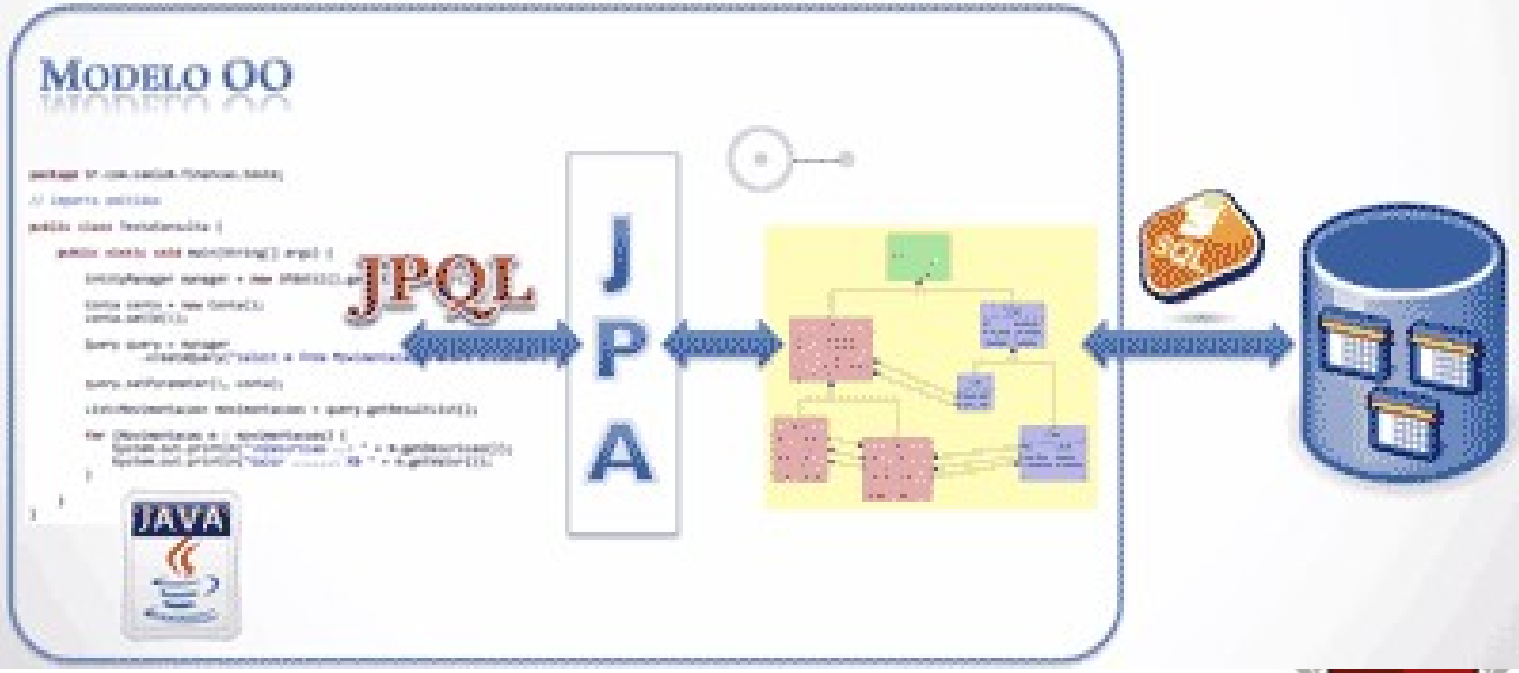


Pesquisas orientadas a objeto

- Com JPA podemos tornar isso mais simples usando apenas o ***Java Persistence Query Language (JPQL)***.
- O JPQL é uma linguagem de consulta, assim como o SQL, porém **orientada a objetos**.
- Isso significa que quando estivermos pesquisando dados, não consideramos nomes de tabelas ou colunas, e sim entidades e seus atributos.
- Através dessa linguagem temos acesso a recursos que o SQL não nos oferece, como polimorfismo e até mesmo maneiras mais simples de buscarmos informações através de relacionamentos.

Consultas com JPQL

Java Persistence Query Language



Criando e executando Queries

- Vamos criar uma classe chamada **TestaConsulta** no pacote br.com.diego.financas.teste
- Faremos primeiro uma consulta simples que retornará todas as movimentações de uma determinada conta
- No método createQuery(..) do EntityManager, passa-se um código JPQL e retorna-se um objeto do tipo javax.persistence.Query.
- Através do objeto Query, podemos solicitar que o resultado seja uma lista de objetos, em nosso caso, de movimentações, através do método **getResultList()**

Criando e executando Queries

- Vamos criar uma classe chamada **TestaConsulta** no pacote br.com.diego.financas.teste
- Faremos primeiro uma consulta simples que retornará todas as movimentações de uma determinada conta
- No método createQuery(..) do EntityManager, passa-se um código JPQL e retorna-se um objeto do tipo javax.persistence.Query.
- Através do objeto Query, podemos solicitar que o resultado seja uma lista de objetos, em nosso caso, de movimentações, através do método **getResultList()**
- **Mas ainda há um problema!**

Criando e executando Queries

- Vamos criar uma classe chamada **TestaConsulta** no pacote br.com.diego.financas.teste
- Faremos primeiro uma consulta simples que retornará todas as movimentações de uma determinada conta
- No método createQuery(..) do EntityManager, passa-se um código JPQL e retorna-se um objeto do tipo javax.persistence.Query.
- Através do objeto Query, podemos solicitar que o resultado seja uma lista de objetos, em nosso caso, de movimentações, através do método **getResultList()**
- **Mas ainda há um problema!**
  - É que a consulta ainda está com um gosto de SQL, estamos comparando o id
  - Solução?

Criando e executando Queries

- Vamos criar uma classe chamada **TestaConsulta** no pacote br.com.diego.financas.teste
- Faremos primeiro uma consulta simples que retornará todas as movimentações de uma determinada conta
- No método createQuery(..) do EntityManager, passa-se um código JPQL e retorna-se um objeto do tipo javax.persistence.Query.
- Através do objeto Query, podemos solicitar que o resultado seja uma lista de objetos, em nosso caso, de movimentações, através do método **getResultList()**
- **Mas ainda há um problema!**
  - É que a consulta ainda está com um gosto de SQL, estamos comparando o id
  - Solução? Passar o objeto como parâmetro!
  - Além do mais, sabemos que ficar concatenando *queries* pode nos levar a problemas grandes, como *SQL Injection*.

Trabalhando com parâmetros na consulta JPQL de maneira segura

- Para resolver ambos os problemas, o JPQL tem uma maneira de definir parâmetros bem similar ao PreparedStatement.
- Na nossa consulta, queremos definir um parâmetro na *query* para indicar que no lugar dele vai entrar a conta recebida de alguma forma.
- No nosso código JPQL vamos remover a concatenação e definir um parâmetro indicado pelo **?1**
- Passaremos este parâmetro através do método setParameter(...) do objeto *query* que recebe a posição do parâmetro e valor que será recebido.

Usando Named Parameters

- A segunda maneira que podemos fazer é ir dando nomes aos parâmetros da *query*.
- Essa segunda maneira é conhecida como *Named Parameter Notation*.
- Agora o parâmetro tem um nome, no nosso caso :pConta e no método setParameter(..)