Roteiro da Aula

Introdução à JPA

Objetivo geral:

※ Conceitar Mapeamento Objeto Relacional;

※ Conhecer os principais recursos fundamentais do JPA (Java Persistence API) na teoria e prática;

※ Elaborar a estrutura básica de um projeto;

Visão geral sobre mapeamento objeto-relacional

Por anos, umas das principais dificuldade na utlização da abordagem orientada a objetos era a comunicação com bancos de dados relacionais.

```
@Override
public List<Seller> findByDepartment(Department dep) {


    PreparedStatement statement = null;
    ResultSet result = null;
    List<Seller> sellers = new ArrayList<Seller>();


    try {

        String sql = "SELECT seller.*,department.Name as DepName "
                + "FROM seller INNER "
                + "JOIN department "
                + "ON seller.DepartmentId = department.Id "
                + "WHERE DepartmentId = ? ORDER BY Name";

        statement = connection.prepareStatement(sql);

        statement.setInt(1, dep.getId());

        result = statement.executeQuery();

        Map<Integer, Department> map = new HashMap<Integer, Department>();

        while(result.next()) {

            Department department = map.get(result.getInt("DepartmentId"));

            if (department == null) {
                department = createDepartment(result);
                map.put(result.getInt("DepartmentId"), department);
            }

            Seller seller = createSeller(result, department);
```

Problemas que precisam ser considerados:

- ※ Contexto de persistência (objetos que estão ou não atrelados a uma conexão em um dado momento);
- ※ Mapa de identidade (cache de objetos já carregados)
- ※ Carregamento tardio (lazy loading);
- ※ Controle de concorrência;
- ※ Outros.

Mapeamento Objeto Relacional (ORM)

Mapeamento objeto relacional (*object-relational mapping*) é uma técnica de programação para "conversão" do modelo relacional para o modelo orientado a objetos.

Em banco de dados, entidades são representadas por tabelas, que possuem colunas que armazenam propriedades de diversos tipos. Uma tabela pode se associar com outras e criar relacionamentos diversos. Já em **uma linguagem orientada a objetos, como Java, entidades são classes**, e objetos dessas classes representam elementos que existem no mundo real.

Por exemplo, um sistema de faturamento possui a classe NotaFiscal, que no mundo real existe e todo mundo já viu alguma pelo menos uma vez, além de possuir uma classe que pode se chamar Imposto. Essas classes são chamadas de classes de domínio do sistema, pois fazem parte do negócio que está sendo desenvolvido.

Em banco de dados, podemos ter as tabelas nota_fiscal e também imposto, mas a estrutura do banco de dados relacional está longe de ser orientado a objetos, e por isso a ORM foi inventada para suprir a necessidade que os desenvolvedores têm de visualizar tudo como classes e objetos para programarem com mais facilidade. Podemos

comparar o modelo relacional com o modelo orientado a objetos conforme a tabela abaixo:

Modelo relacional	Modelo OO
Tabela	Classe
Linha	Objeto
Coluna	Atributo
-	Método
Chave estrangeira	Associação

Essa comparação é feita em todo o tempo quando estamos desenvolvendo usando algum mecanismo de ORM. O mapeamento é feito usando metadados que descrevem a relação do modelo
relacional do banco de dados e o modelo orientado a objetos.

Java Persistence API (JPA)

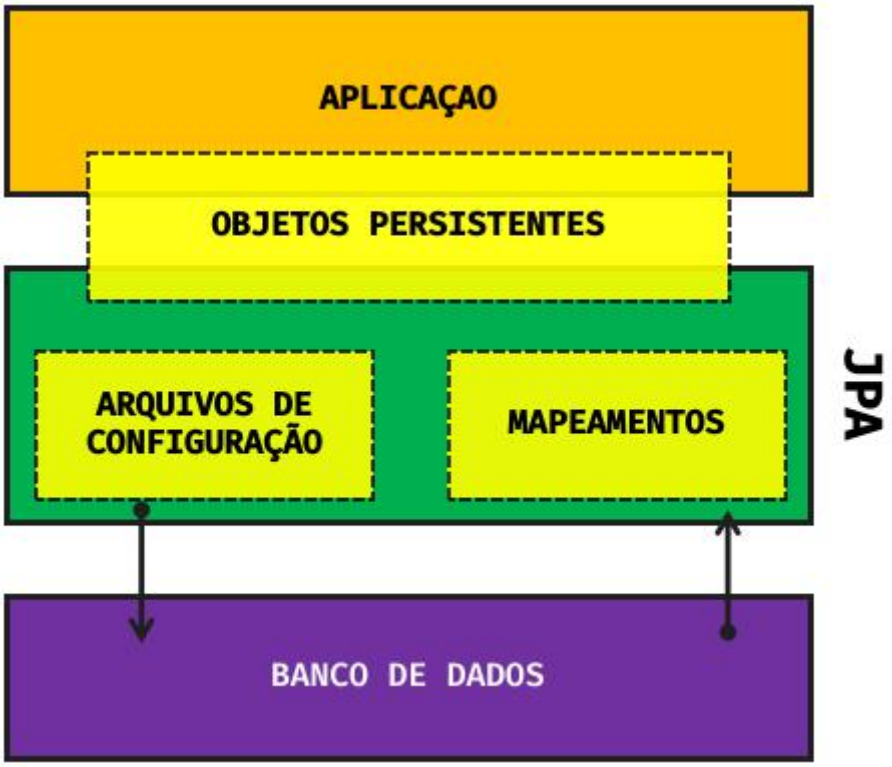
A Java Persistence API (JPA) ou *Jakarta Persistence* (novo nome adotado pelo JakartaEE) é uma especificação que oferece uma API de mapeamento objeto-relacional e persistência de dados.

JPA é “apenas” uma especificação (JSR 338):
http://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf

O JPA até que possui um artefato JAR com algumas classes e interfaces no pacote javax.persistence, mas só isso não resolve nada. Não é a implementação ORM de fato.

Para trabalhar com JPA é preciso incluir no projeto uma implementação da API. O mais famoso é o ***Hibernate***.

Arquitetura de uma aplicação que utiliza JPA:



Principais classes da API

EntityManager https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html

Um objeto EntityManager encapsula uma conexão com a base de dados e serve para efetuar operações de **acesso a dados** (inserção, remoção, deleção, atualização) em **entidades** (clientes, produtos, pedidos, etc.) por ele **monitoradas** em um mesmo **contexto de persistência**.

Escopo: *tipicamente mantem-se uma instância única de EntityManager* ***para cada thread do sistema*** *(no caso de aplicações web, para cada requisição ao sistema).*

EntityManagerFactory
https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManagerFactory.html

Um objeto EntityManagerFactory é utilizado para instanciar objetos EntityManager.
Escopo: *tipicamente mantem-se uma instância única de EntityManagerFactory para toda aplicação.*

Primeiros passos com JPA

Criando e configurando o projeto

Crie um projeto Web. Quando o projeto for criado, um arquivo chamado ***pom.xml*** será criado no diretório raiz do projeto.

Esse arquivo contém informações sobre o projeto e detalhes de configurações usadas para o Maven fazer build do projeto.

Para começar a trabalhar com JPA, precisamos adicionar algumas dependências no pom.xml. Abra o arquivo pom.xml, encontre a tag <*dependencies*> e adicione as duas dependências, conforme o código abaixo:

```
<dependencies>
    <dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.2.0.CR3</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.20</version>
    </dependency>

</dependencies>
```

O Maven baixa automaticamente os JARs das dependências pela internet, do repositório central. A dependência hibernate-core adiciona o Hibernate no nosso projeto, que é a implementação JPA que nós vamos usar.

Ao adicionar a implementação do Hibernate como dependência, estamos automaticamente adicionando também o JPA, que é uma dependência do Hibernate (o Hibernate depende do JPA, portanto "carrega" junto essa dependência). Esse tipo de dependência é chamada de dependência transitiva.

** Verifique as dependências baixadas na pasta External Libraries;*

Criando o Domain Model

Em nosso projeto, queremos gravar e consultar informações de veículos de uma loja no banco de dados. Antes de qualquer coisa, precisamos criar nosso modelo de domínio para o negócio em questão.

Checklist:

- ※ Crie um pacote chamado 'domain";
- ※ Dentro do pacote domain, crie outro pacote chamado "entities";
- ※ Dentro do pacote entities crie uma classe chamada Veiculos;
- ※ Implementando o equals() e hashCode() para Veiculos;

```java
public class Veiculo {

    private Long codigo;
    private String fabricante;
    private String modelo;
    private Integer anoFabricacao;
    private Integer anoModelo;
    private BigDecimal valor;

}
```

O atributo identificador (chamado de codigo) é referente à chave primária da tabela de veículos no banco de dados. Se existirem duas instâncias de Veiculo com o mesmo identificador, eles representam a mesma linha no banco de dados.

As classes de entidades podem seguir o estilo de JavaBeans, com métodos getters e setters. É obrigatório que estas classes possuam um construtor sem argumentos.

Qual é a real importância dos métodos equals e hashCode?

Para que os objetos das entidades sejam diferenciados uns de outros, precisamos implementar os métodos equals() e hashCode(). No banco de dados, as chaves primárias diferenciam registros distintos. Quando mapeamos uma entidade para uma tabela, devemos criar os métodos equals() e hashCode(), levando em consideração a forma em que os registros são diferenciados no banco de dados.

Entender os motivos por trás da implementação desses métodos é muito importante. **Faça uma pesquisa acerca do assunto.**

Mapeamento básico

Para que o mapeamento objeto-relacional funcione, precisamos informar à implementação do JPA mais detalhes sobre como objetos da classe Veiculo devem se tornar persistentes, ou seja, como instâncias dessa classe podem ser gravadas e consultadas no banco de dados. Para isso, devemos anotar os getters ou os atributos, além da própria classe, com anotações do JPA.

```
@Entity
public class Veiculo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column
    private String fabricante;
    private String modelo;
    private Integer anoFabricacao;
    private Integer anoModelo;
    private BigDecimal valor;
}
```

Você deve ter percebido que as anotações foram importadas do pacote jakarta.persistence. Dentro desse pacote estão todas as anotações padronizadas pela JPA.

A ***anotação @Entity*** diz que a classe é uma entidade JPA, que representa uma tabela do banco de dados.

Já as **anotações nos atributos** configuram a ***relação com as colunas da tabela do banco de dados***.

A ***anotação @Id*** é usada para declarar o identificador da entidade, ou seja, representa a chave primária na tabela do banco de dados. A ***anotação @GeneratedValue*** especifica que um valor será gerado automaticamente para este atributo. Definimos a estratégia de geração do identificador através da propriedade strategy com o valor GenerationType.IDENTITY.

A estratégia IDENTITY especifica que o valor será auto-incrementado pela própria coluna do banco de dados. Ou seja, ao inserir um novo registro, esperamos que o próprio banco de dados tome conta do incremento para o código do veículo.

A ***anotação @Column*** especifica que a propriedade da classe representa uma coluna na tabela do banco de dados. Propositalmente, anotamos apenas uma propriedade com @Column, mas isso não quer dizer que apenas a propriedade anotada está mapeada. Ao omitir a anotação nas outras propriedades, o JPA faz o mapeamento automaticamente, o que significa que todas as propriedades da classe estão mapeadas para colunas.

Checklist

※ Inclua os mapeamentos nas entidades (classes de domínio) do seu projeto.

O arquivo persistence.xml

Material de apoio: persistence.xml

O ***persistence.xml*** é um arquivo de configuração padrão da JPA. Ele deve ser criado no diretório META-INF da aplicação ou do módulo que contém as classes de entidade. O arquivo persistence.xml define unidades de persistência, conhecidas como persistence units.

Checklist:

- ※ Crie uma pasta no pacote src/main/resources chamada "**META-INF"**;
- ※ Dentro dessa pasta crie um arquivo chamado persistence.xml;
- ※ Usando o modelo passado no material de apoio, copie o conteúdo, cole e faça as devidas atulizações no arquivo;

Lembre-se do nome da unidade de persistência, pois é por meio dela que teremos acesso a essas configurações.

Gerando as tabelas do banco de dados

Como ainda não temos a tabela representada pela classe Veiculo no banco de dados, precisamos criá-la.

O JPA pode fazer isso pra gente, graças à propriedade *jakarta.persistence.schemageneration.database.action* com valor ***drop-and-create***, que incluímos no arquivo persistence.xml.

Precisamos apenas obter uma instância de EntityManagerFactory para dar um start no mecanismo do JPA. **Assim, todas as tabelas mapeadas pelas entidades serão criadas.**

```
public class AppTest {

    public static void main(String[] args) {

        EntityManagerFactory entityManagerFactory;
        entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "ifpr_aula_jpa");

    }
}
```

O parâmetro do método ***createEntityManagerFactory*** deve ser o mesmo nome que informamos no atributo name da tag persistence-unit, no arquivo persistence.xml. Ao executar o código, a tabela Veiculo é criada. ***Observe a saída na console do Intellij.***

Pré populando nossa base de dados:

Para fins de testes e experimentação é conveniente que nossa base de dados já venha populada, isto é, preenchida com alguns registros.

Checklist:

- ※ Dentro da pasta resources, crie um arquivo chamado de dados-iniciais.sql;
- ※ Usando este link como base, crie Veiculos para serrem inseridos na inicialização do seu projeto;
- ※ Adicione a seguinte propriedade ao arquivo persistence.xml:

```
<property name="jakarta.persistence.sql-load-script-source" value="META-INF/dados-iniciais.sql"/>
```

Definindo detalhes físicos de tabelas

A estrutura da tabela que foi criada no banco de dados é bastante simples (acesse seu Mysql Workbench, encontre a tabela Veiculo e clique em **"Alter Table.."). ** Observe que todas as propriedades da classe foram mapeadas automaticamente, mesmo sem incluir qualquer anotação JPA.

Com exceção da chave primária, todas as colunas aceitam valores nulos e as colunas fabricante e modelo aceitam até 255 caracteres. Podemos definir melhor estes detalhes físicos no mapeamento de nossa entidade. Além disso, queremos que nomes de colunas usem underscore na separação de palavras, que o nome da tabela seja tab_veiculo e a precisão da coluna valor seja 10, com 2 casas decimais.

```java
@Entity(name = "veiculo")
public class Veiculo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long codigo;

    @Column(length = 60, nullable = false)
    private String fabricante;

    @Size(min = 2, max = 30, message = "About Me must be between 10 and 200 characters")
    private String modelo;

    @Column(name = "ano_fabricacao", nullable = false)
    private Integer anoFabricacao;

    @Column(name = "ano_modelo", nullable = false)
    private Integer anoModelo;

    @Column(precision = 10, scale = 2, nullable = true) //até 10 digitos com duas casas decimais
    private BigDecimal valor;
```

Como tivemos alteração no nome da tabela e de algumas colunas, então, antes de testar, é preciso atualizar o nosso arquivo dados-iniciais.sql.

Criando EntityManager

Os sistemas que usam JPA precisam de apenas uma instância de EntityManagerFactory, que pode ser criada durante a inicialização da aplicação. Esta única instância pode ser usada por qualquer código que queira obter um EntityManager.

Um EntityManager é responsável por gerenciar entidades no contexto de persistência **(que você aprenderá mais a frente)**. Através dos métodos dessa interface, é possível persistir, pesquisar e excluir objetos do banco de dados.

Seguindo a recomendação de que exista apenas uma instância de EntityManagerFactory na aplicação, utlizaremos uma classe para acesso e compartilhamento da instância dessa classe.

```
public class ConnectionFactory {

    private static EntityManagerFactory entityManagerFactory = null;

    static {
        entityManagerFactory = Persistence.createEntityManagerFactory( persistenceUnitName: "ifpr_aula_jpa");
    }

    public static EntityManager getEntityManager(){

        return entityManagerFactory.createEntityManager();
    }

    public static void closeEntityManagerFactory(EntityManagerFactory emFactory) { emFactory.close(); }

    public static void closeEntityManager(EntityManager em) { em.close(); }

}
```

Criamos um bloco estático para inicializar a fábrica de Entity Manager. Isso ocorrerá apenas uma vez, no carregamento da classe. ***É similar ao udo do padrão Singleton, usado na aula de Acesso ao bando de dados***. Agora, sempre que precisarmos de uma EntityManager, podemos chamar:

EntityManager manager = ConnectionFactory.getEntityManager();

Persistindo objetos

```
EntityManager entityManager = ConnectionFactory.getEntityManager();
entityManager.getTransaction().begin();

Veiculo v1 = new Veiculo();
v1.setModelo("Fusca");
v1.setFabricante("VW");
v1.setAnoFabricacao(1980);
v1.setAnoModelo(1981);
v1.setValor(new BigDecimal( val: "5000"));

entityManager.persist( o: v1);

entityManager.getTransaction().commit();
```

Consulte os dados da tabela tab_veiculo usando qualquer ferramenta de gerenciamento do seu banco de dados e veja que as informações que definimos nos atributos da instância do veículo, através de métodos setters, foram armazenadas.

Entendendo cada linha:

O código abaixo obtém um EntityManager, que é responsável por gerenciar o ciclo de vida das entidades.

```
EntityManager entityManager= ConnectionFactory.getEntityManager();
```

Agora iniciamos uma nova transação.

```
entityManager.getTransaction().begin( );
```

Instanciamos um novo veículo e atribuímos alguns valores, chamando os métodos setters.

```
Veiculo v1 = new Veiculo();
v1.setModelo("Fusca");
v1.setFabricante("VW");
v1.setAnoFabricacao(1980);
v1.setAnoModelo(1981);
v1.setValor(new BigDecimal( val: "5000"));
```

Executamos o método persist, passando a instância do veículo como parâmetro. Isso fará com que o JPA insira o objeto no banco de dados.

Não informamos o código do veículo, porque ele será obtido automaticamente através do auto-increment do MySQL.

```
entityManager.persist(veiculo);
```

Agora fazemos commit da transação, para efetivar a inserção do veículo no banco de dados.

```
entityManager.getTransaction().commit();
```

Não precisamos escrever nenhum código SQL e JDBC. Trabalhamos apenas com classes, objetos e a API de JPA. **É claro que, no final das contas, SQL e JDBC continuam sendo usados por baixo dos panos**, pela implementação do JPA.

Buscando objetos pelo identificador

Podemos recuperar objetos através do identificador (chave primária) da entidade. O código abaixo busca um veículo com o código igual a 1. Agora podemos executar o código abaixo, que faz uma consulta por um veículo de identificador 1. Podemos realizar consultas ao banco de dados utlizando os métodos **find** e **getReference;**

```
Veiculo veiculo = entityManager.find(Veiculo.class, 1L);
Veiculo veiculo = entityManager.getReference(Veiculo.class, 1L);
```

Embora tenham resultados parecidos, existe uma diferença importante no modo em que cada método obtém o objeto: enquanto o método find busca o objeto imediatamente no banco de dados, o método getReference só executa o SQL quando o objeto for usado pela primeira vez, ou seja, quando invocamos um método getter da instância.

Listando objetos

Para realizar consultas complexas, ainda existe a possibilidade de se utlizar a linguagem SQL. O JPA também traz uma variação da linguagem SQL, o JPQL (*Java Persistence Query Language*).

A JPQL é uma extensão da SQL, porém com características da orientação a objetos. Com essa linguagem, não referenciamos tabelas do banco de dados, mas apenas entidades de nosso modelo.

Quando fazemos pesquisas em objetos, não precisamos selecionar as colunas do banco de dados, como é o caso da SQL. O código em SQL a seguir:

```
select * from veiculo
```

Fica da seguinte forma em JPQL:

```
select v from Veiculo v
```

A sintaxe acima em JPQL significa que queremos buscar os objetos da entidade Veiculo.

```
Query query = entityManager.createQuery("select v from Veiculo v");
List veiculos = query.getResultList();
```

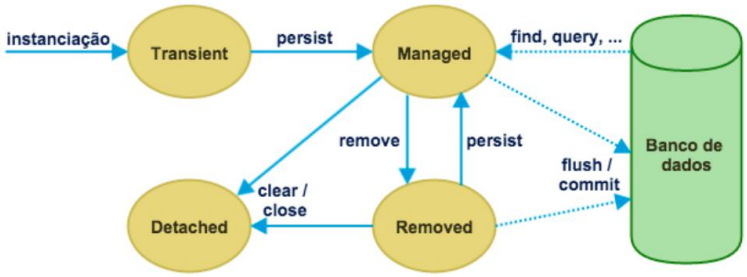
Checklist:

※ Implemente um método que atualize registros do banco de dados;

※ Implemente um método que excluar objetos do banco de dados;

Estados e ciclo de vida e Contexto de Persistência

Objetos de entidades são instâncias de classes mapeadas usando JPA, que ficam na memória e representam registros do banco de dados. Essas instâncias possuem um ciclo de vida, que é gerenciado pelo JPA. Os estados do ciclo de vida das entidades são: *transient (ou new)*, *managed*, *detached* e *removed*.



Contexto de persistência

O contexto de persistência é uma coleção de objetos gerenciados por um EntityManager. Se uma entidade é pesquisada, mas ela já existe no contexto de persistência, o objeto existente é retornado sem acessar o banco de dados. **Esse recurso é chamado de cache de primeiro nível.**

É importante ressaltar que uma mesma entidade pode ser representada por diferentes objetos na memória, desde que seja em diferentes instâncias de EntityManagers

```
Veiculo veiculo1 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela primeira vez...");

Veiculo veiculo2 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela segunda vez...");

System.out.println("Mesmo veículo? " + (veiculo1 == veiculo2));
```

O método ***contains*** de EntityManager verifica se o objeto está sendo gerenciado pelo contexto de persistência do EntityManager. O método ***detach*** para de gerenciar a entidade no contexto de persistência, colocando ela no estado detached.

```
Veiculo veiculo1 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela primeira vez...");

System.out.println("Gerenciado? " + manager.contains(veiculo1));
manager.detach(veiculo1);
System.out.println("E agora? " + manager.contains(veiculo1));

Veiculo veiculo2 = manager.find(Veiculo.class, 2L);
System.out.println("Buscou veiculo pela segunda vez...");

System.out.println("Mesmo veículo? " + (veiculo1 == veiculo2));
```