



**CES/JF**  
Centro de Ensino Superior  
de Juiz de Fora

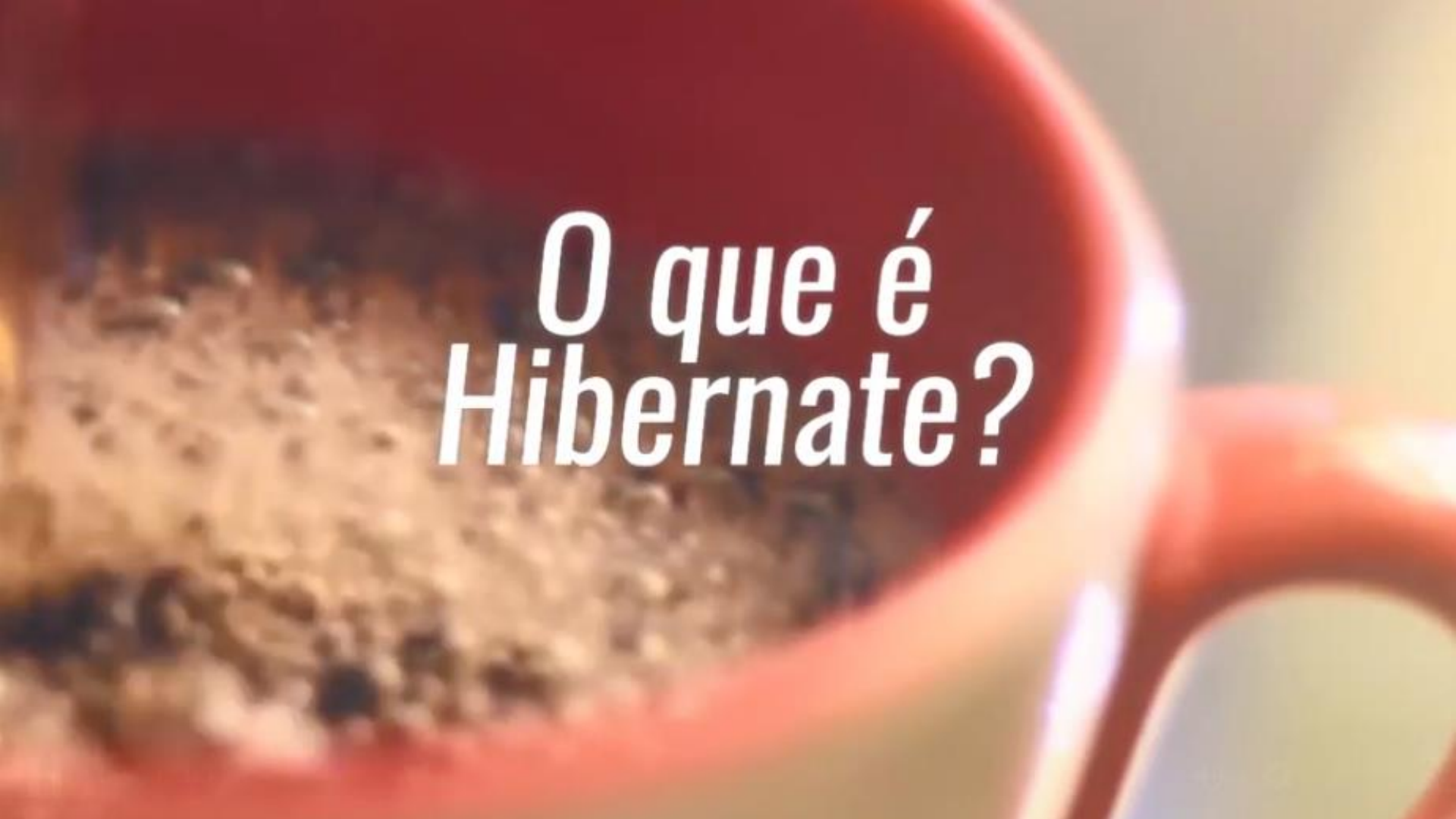
MANTENEDORA



**SVD - ESDEVA**

# Persistência – JPA / Hibernate

Prof. Christien Lana Rachid  
2019/1

A close-up photograph of a red plastic bowl filled with dark brown soil. Numerous small, dark, oval-shaped seeds are scattered throughout the soil. The text "O que é Hibernante?" is overlaid in white, italicized font on the right side of the bowl.

*O que é  
Hibernante?*

# Hibernate e a API JDBC

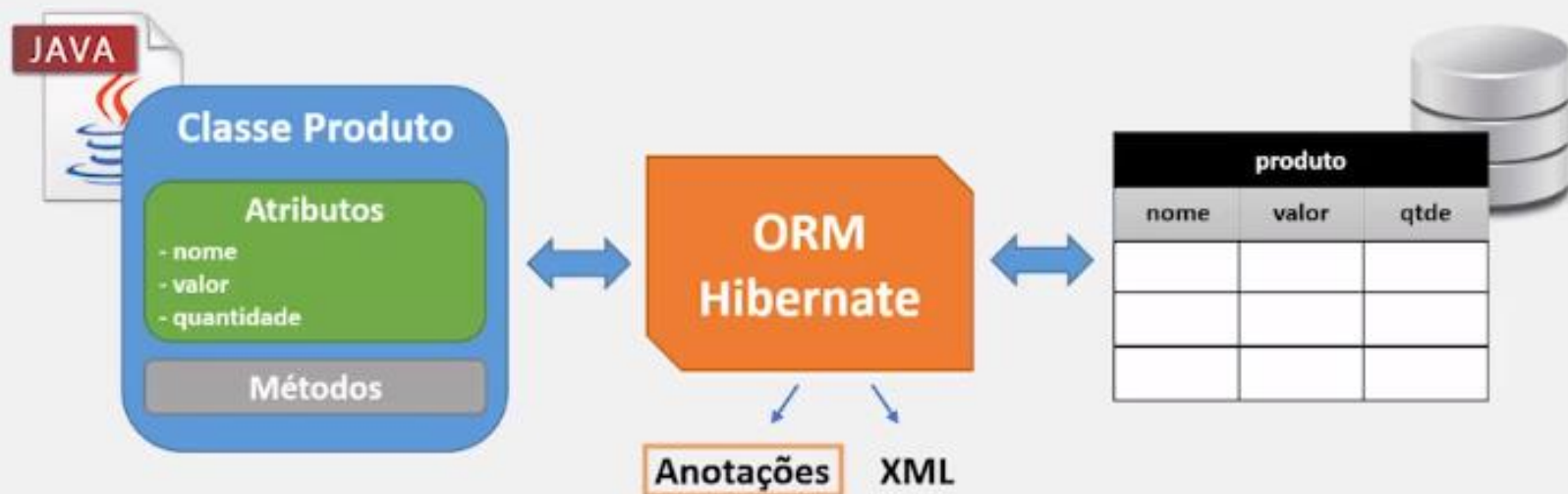
## JDBC

- Preocupação com a conversão dos dados (Relacional x OO)
- Escrita de código SQL
- Escrita de mais linhas de código

# Hibernate | Introdução

- ❑ Com a popularização do Java no meio corporativo, observou-se o grande esforço dedicado à criação de SQL e manipulação JDBC.
- ❑ Além da produtividade foram observados outros problemas:
  - ❑ Diferença entre a sintaxe SQL de SGBDs diferentes.
  - ❑ Desafio no mapeamento objeto-relacional

# Hibernate e o ORM



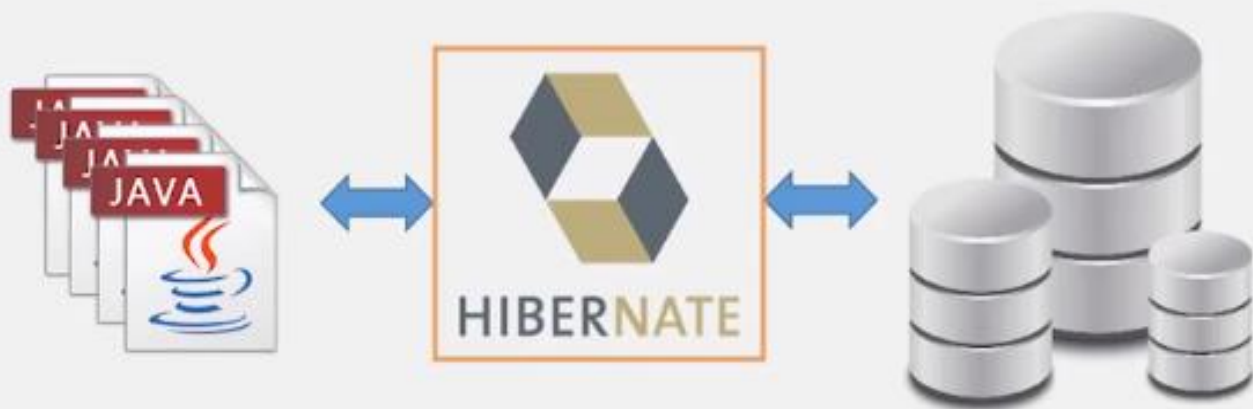
# Hibernate | Introdução

- ❑ Diferentes alternativas para solucionar esse problema surgiram.
- ❑ Entre elas o Hibernate se popularizou, se tornando, desde então, líder de mercado.
- ❑ O hibernate é um framework ORM (**M**apeamento **O**bjeto **R**elacional), e busca resolver o que conhecemos como impedância objeto (representação seguindo princípios OO) relacional (representação normalizada).



# HIBERNATE

# Hibernate e a API JPA



# Arquitetura com o Hibernate

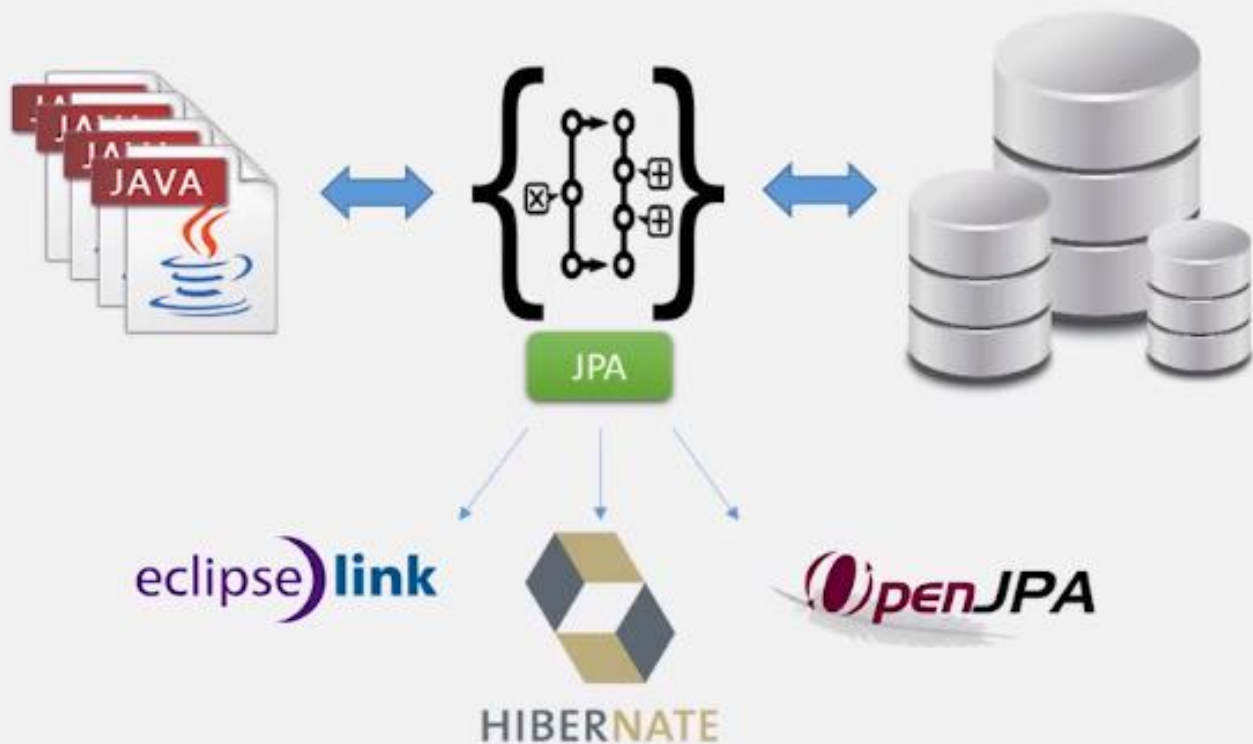




## JPA/Hibernate | Introdução

- ❑ Por sua popularidade e forte presença em diversos projetos Java, uma proposta de especificação de um framework inspirado no Hibernate foi proposto.
- ❑ O JPA, Java Persistence API, define uma série de classes, interfaces e padrões para a criação de frameworks ORM.
- ❑ O Hibernate é um dos frameworks que implementam o padrão JPA, assim como o EclipseLink (referência) e o OpenJPA.

# Hibernate e a API JPA

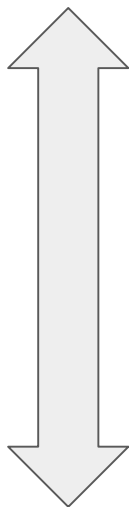


# JPA/Hibernate | Introdução

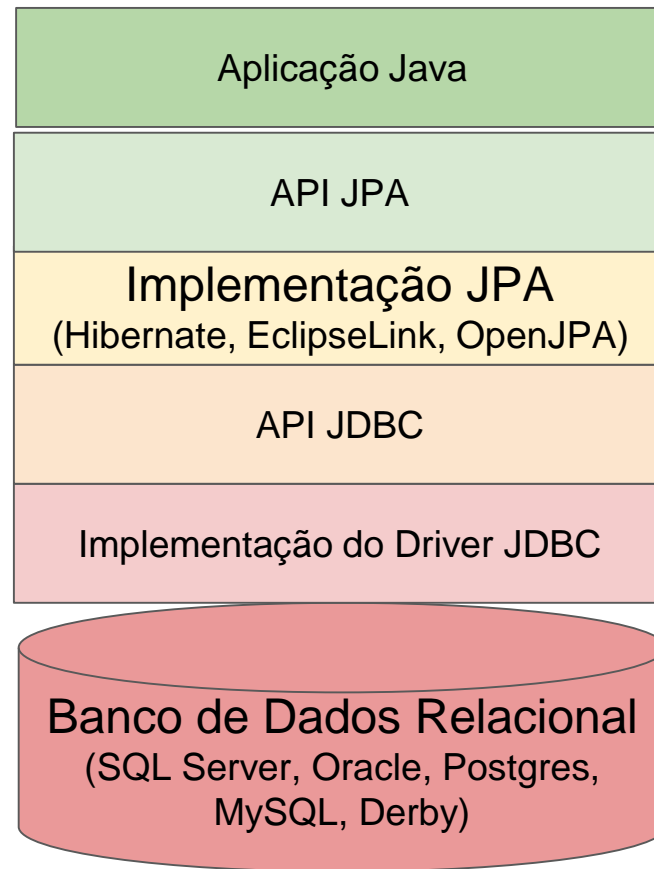
- ❑ Vamos focar no estudo do JPA, considerando o Hibernate apenas uma implementação do mesmo.
  - ❑ O Hibernate, assim como outras implementações, possui recursos específicos que não estão no padrão. Quando for utilizado algum recurso específico, este será destacado.
  - ❑ Apenas em situações muito específicas deve-se utilizar algum recurso exclusivo da implementação do JPA.

# JPA/Hibernate | Arquitetura

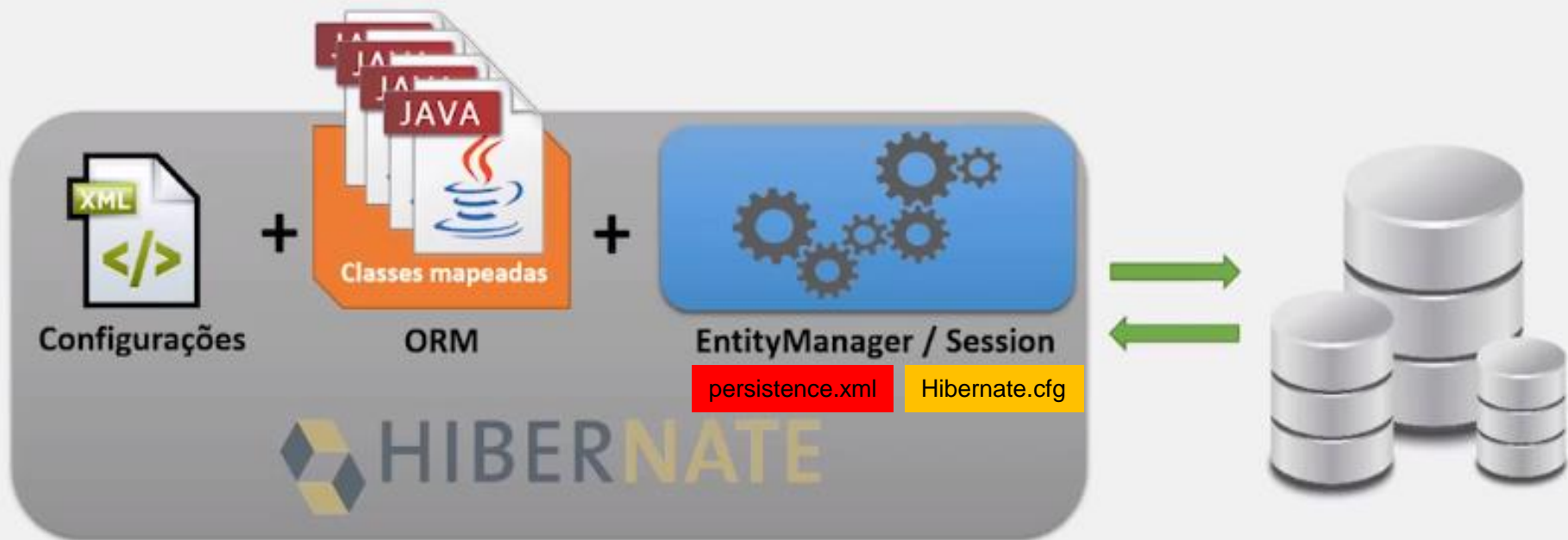
Modelo Orientado  
à Objetos



Modelo Relacional



# Programando com o Hibernate



# JPA/Hibernate | Unidade de Persistência

- ❑ Para utilizar o JPA, o primeiro passo é definir uma unidade de persistência (***persistent unit***).
- ❑ Deve ser definida no arquivo `persistence.xml`
  - ❑ Que deve estar na pasta META-INF, na raiz do ***classpath***.
  - ❑ `src/META-INF/persistence.xml`
- ❑ Uma unidade de persistência define:
  - ❑ As informações de conexão JDBC com o banco (URL, Driver...)
  - ❑ As informações necessárias para a implementação (Hibernate...)
  - ❑ Atenção, todas as classes necessárias para o provider e para o JDBC devem estar no ***classpath***.

# JPA/Hibernate | Unidade de Persistência

Definição da unidade de persistência. Uma aplicação java pode ter várias unidades de persistência.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.1"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence-2.1.xsd">
  <persistence-unit name="estacionamento_unit">
    <description> Hibernate JPA Estacionamento</description>
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:derby:estacionamento;create=true"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value="root"/>
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Definição da implementação do JPA que deve ser utilizada nessa unidade de persistência. Nesse caso, a classe que define que utilizaremos o Hibernate é: **org.hibernate.jpa.HibernatePersistenceProvider**

# JPA/Hibernate | Unidade de Persistência

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.1"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence"
<persistence-unit name="estacionamento_unit">
  <description> Hibernate JPA Estacionamento</description>
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <properties>
    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby:estacionamento;create=true"/>
    <property name="javax.persistence.jdbc.user" value="root"/>
    <property name="javax.persistence.jdbc.password" value="root"/>
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.format_sql" value="true" />
    <property name="hibernate.hbm2ddl.auto" value="create" />
  </properties>
</persistence-unit>
</persistence>
```

Configurações padrão do JPA para definição do JDBC utilizado para acessar o banco de dados.

As propriedades do Hibernate são para que ele exiba no console as queries realizadas (**show\_sql**), formatadas (**format\_sql**) e para que ele recrie as tabelas do banco a cada execução (**hbm2ddl.auto**).



# Hibernate | Unidade de Persistência

- ❑ A partir da unidade de persistência, acessamos outras classes essenciais para a JPA.
- ❑ A definição de unidades de persistência desacopla detalhes específicos do provider das definições da especificação JPA.
- ❑ Se apenas o padrão da API for utilizado, é possível substituir o provider sem impactar outras partes da aplicação.
  - ❑ Na prática é muito difícil de fazer

# Hibernate | Entidades

- ❑ Uma **entidade** é um objeto que pode ser persistido no banco de dados.
- ❑ Essas classes possuem **anotações** especiais indicando como devem ser persistidas em um banco de dados relacional.
  - ❑ A partir dessas informações o Hibernate sabe como fazer o mapeamento objeto relacional entre as tabelas e colunas do banco de dados e os objetos e atributos em Java.
- ❑ Entidades devem possuir identificadores únicos, e existem independentemente dos valores de seus atributos.
  - ❑ Estão relacionados à uma linha de uma tabela no banco de dados.

# Hibernate | Entidades | Regras

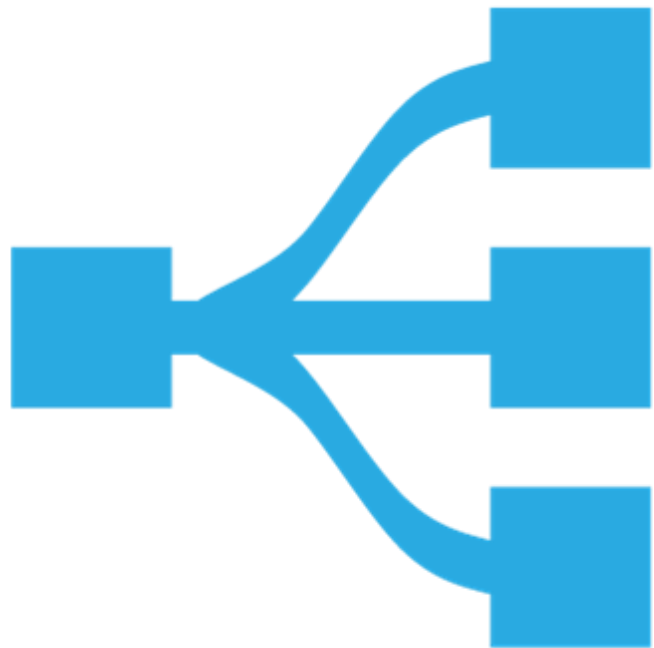
- ❑ Devem estar anotadas com **@Entity**
- ❑ Deve possuir um construtor sem argumentos **public** ou **protected** (Pode possuir outros).
- ❑ Não pode ser uma **inner-class**, **enum** ou **interface**.
- ❑ Não pode ser **final** ou ter membros mapeados **final**.
- ❑ Os atributos devem estar encapsulados, acessíveis somente a partir da própria entidade e externamente através de getters, setters e outros métodos de negócio.

# Hibernate | Entidades | Regras

- ❑ Tipicamente as entidades são objetos do **domínio** da aplicação, isto é, do universo em que a aplicação deve atuar.
- ❑ Entidades podem herdar de classes que não são entidades, e classes que não são entidades podem herdar de entidades.
  - ❑ O JPA suporta 3 formas de mapeamento de herança entre atividades.
- ❑ Por razões históricas, o JPA/Hibernate suportam a definição de entidades sem um id, no entanto essa é um recursos ultrapassado que deve ser removido nas próximas versões.

# Hibernate | Mapeamento

- ❑ Existem dois tipos de mapeamento.
  - ❑ Mapeamento de valores.
  - ❑ Mapeamento de entidades.



# Hibernate | Mapeamento | Valores

- ❑ Diversos tipos de dados padrão do Java podem ser mapeados diretamente para o banco de dados em forma de coluna:
  - ❑ Tipos primitivos (int, char, byte) e suas classes correspondentes (Integer, Character, Byte), assim como Strings.
  - ❑ As classes: java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp.
  - ❑ Os Enums.
  - ❑ Tipos de data do Java 8.
  - ❑ Tipos definidos como valores por meio de converters ( que convertem um objeto em um valor de coluna e vice versa.)

# Hibernate | Mapeamento | Entidades

- ❑ Definidos basicamente através da anotação `@Entity` e com uma coluna definida como `@Id`.

```
@Entity(name = "Product")
public class Product {

    @Id
    private Integer id;

    private String sku;

    private String name;

    private String description;

    // getters e setters omitidos.
}
```

# Hibernate | Mapeamento | Entidades

- ❑ O nome da tabela e os nomes da coluna são definidos por convenção. No entanto o nome da tabela pode ser definido pela anotação `@Table` e o nome da coluna pode ser definido pela anotação `@Column`. Além disso, o Hibernate oferece outros padrões e formas de se definir um padrão particular.

```
@Entity(name = "Product")
@Table(name = "Products")
public class Product {

    @Id
    private Integer id;

    private String name;

    @Column( name = "NOTES" )
    private String description;
}
```



# Hibernate | Mapeamento | Entidades

- ❑ A anotação `@Column` pode ser usada ainda para definir outras informações de mapeamento.

```
@Entity(name = "Product")
@Table(name = "Products")
public class Product {
    @Id
    private Integer id;

    private String name;
```

Para valores decimais, podemos definir o número de dígitos (**precision**) e casas decimais (**scale**). Por exemplo, o numero 123456,78 possui **precision=8** e **scale=2**

Podemos definir se uma coluna aceita valores nulos (**nullable**) e se aceita valores repetidos entre linhas (**unique**). Por padrão ambos são **false**.

```
@Column( name = "US_PRICE", precision=8, scale=2, nullable=false, unique=false)
private Double price;
```

```
@Column( name = "NOTES", columnDefinition = "BIGTEXT", length=1500 )
```

Podemos ainda definir o tipo específico que uma coluna deve ter no banco de dados por meio do **columnDefinition**, e o tamanho de um tipo no banco por meio do atributo **length**

# Hibernate | Mapeamento | Entidades

- ❑ Para atributos compostos, podemos utilizar classes diferentes com a anotação **@Embeddable**.

```
@Entity(name = "Contact")
public static class Contact {

    @Id
    private Integer id;

    private Name name;

    private String notes;

    private URL website;

    private boolean starred;
    //Getters e setters omitidos
}
```

```
@Embeddable
public class Name {

    private String first;

    private String middle;

    private String last;

    // Getters/setters
    // omitidos
}
```

```
create table Contact (
    id integer not null,
    first varchar(255),
    last varchar(255),
    middle varchar(255),
    notes varchar(255),
    starred boolean not
null,
    website varchar(255),
    primary key (id)
)
```

# Hibernate | Mapeamento | Entidades | Id

- ❑ Para definir um `@Id` autogerado, diferentes estratégias podem ser utilizadas por meio da anotação `@GeneratedValue`.

## @Entity

```
public class EntityWithAutoId2 {  
    @Id @GeneratedValue(strategy=GenerationType.AUTO) long id;  
}
```

A estratégia **GenerationType.AUTO** delega para o provider (Hibernate) a responsabilidade de escolher qual o tipo de geração deve ser utilizado. No caso dos principais provider esse tipo é **GenerationType.IDENTITY**.

# Hibernate | Mapeamento | Entidades | Id

- ❑ Para gerar por meio de uma sequência utiliza-se o tipo de geração `GenerationType.SEQUENCE` em conjunto com o `@GeneratedValue` e `@SequenceGenerator`

@Entity

@SequenceGenerator(name="my\_generator", sequenceName = "my\_seq")

```
public class EntityWithAutoId2 {
```

```
    @Id @GeneratedValue(strategy=GenerationType.SEQUENCE, name="my_seq")
```

```
    private long id;
```

```
}
```

# Hibernate | Mapeamento | Entidades

- ☐ Além disso é possível mapear entidades diferentes usando a anotação **@Converter**. Porém, detalhes da implementação do mesmo fogem do escopo da disciplina.
- ☐ Existem recursos para gerar automaticamente atributos (alterado por, criado por).
- ☐ Recursos para adicionar resultados de fórmulas a campos.
- ☐ Manipular dados para serem serializados e de-serializados do banco de dados.
- ☐ Para gerar e obter IDs.

# Hibernate | Mapeamento | Associações

- ❑ As associações são relações entre diferentes entidades, que podem ser:
  - ❑ Um para muitos.
  - ❑ Muitos para Um.
  - ❑ Muitos para Muitos.
- ❑ Deve-se ficar atento à modelagem de relacionamentos pois a mesma tem sérias implicações em relação à performance das operações sobre o modelo.

# Hibernate | Mapeamento | Associações | @ManyToOne

- ❑ O tipo mais comum de relacionamento é a associação de um “filho” com seu “pai”, é o equivalente ao relacionamento por meio de foreign key em bancos relacionais.

```
@Entity(name = "Person")
public static class Person {
    @Id
    @GeneratedValue
    private Long id;

    public Person() {
    }
}
```

```
@Entity(name = "Phone")
public static class Phone {
    @Id
    @GeneratedValue
    private Long id;

    @Column(name = "`number`")
    private String number;

    @ManyToOne
    @JoinColumn(name = "person_id",
        foreignKey =
    @ForeignKey(name = "PERSON_ID_FK")
    )
    private Person person;
}
```

```
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)

CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
)

ALTER TABLE Phone
ADD CONSTRAINT PERSON_ID_FK
FOREIGN KEY (person_id)
REFERENCES Person
```

# Hibernate | Mapeamento | Associações | @ManyToOne

## ❑ Ciclo de vida de uma associação @ManyToOne

```
Person person = new Person();
entityManager.persist( person );

Phone phone = new Phone( "123-456-7890" );
phone.setPerson( person );
entityManager.persist( phone );

entityManager.flush();
phone.setPerson( null );
```

```
INSERT INTO Person ( id )
VALUES ( 1 )

INSERT INTO Phone ( number, person_id, id )
VALUES ( '123-456-7890', 1, 2 )

UPDATE Phone
SET    number = '123-456-7890',
       person_id = NULL
WHERE id = 2
```



# Hibernate | Mapeamento | Associações | @OneToMany

- ❑ Liga uma entidade “pai” a um ou mais filhos que se associam a ele por meio de uma chave estrangeira.

```
@Entity(name = "Person")
public static class Person {

    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "person",
        cascade = CascadeType.ALL,
        orphanRemoval = true)
    private List<Phone> phones;
}
```

```
@Entity(name = "Phone")
public static class Phone {

    @Id
    @GeneratedValue
    private Long id;

    @NaturalId
    @Column(name = "`number`",
        unique = true)
    private String number;

    @ManyToOne
    private Person person;
}
```

```
CREATE TABLE Person (
    id BIGINT NOT NULL ,
    PRIMARY KEY ( id )
)
CREATE TABLE Phone (
    id BIGINT NOT NULL ,
    number VARCHAR(255) ,
    person_id BIGINT ,
    PRIMARY KEY ( id )
)
ALTER TABLE Phone
ADD CONSTRAINT
UK_I329ab0g4c1t78onljnxmbnp6
UNIQUE (number)

ALTER TABLE Phone
ADD CONSTRAINT
FKmw13yfsjypiiq0i1osdkaeqpg
FOREIGN KEY (person_id)
REFERENCES Person
```

# Hibernate | Mapeamento | Associações | @OneToMany

❑ Ciclo de vida de um @OneToMany.

```
Person person = new Person();  
  
Phone phone1 = new Phone( "123-456-7890");  
Phone phone2 = new Phone( "321-654-0987");  
  
person.addPhone( phone1 );  
person.addPhone( phone2 );  
entityManager.persist( person );  
entityManager.flush();  
  
person.removePhone( phone1 );
```

```
INSERT INTO Phone  
        ( number, person_id, id )  
VALUES ( '123-456-7890', NULL, 2 )  
  
INSERT INTO Phone  
        ( number, person_id, id )  
VALUES ( '321-654-0987', NULL, 3 )  
  
DELETE FROM Phone  
WHERE   id = 2
```