

# JAVA SPRING DATA

---

Prof. Júlio Machado

[julio.machado@pucrs.br](mailto:julio.machado@pucrs.br)

# JPA

---

## Introdução e Mapeamentos


# JPA

- Jakarta Persistence API é uma especificação do JavaEE (Java Enterprise Edition) utilizada para implementar a camada de persistência usando mapeamento objeto-relacional.
- É uma implementação do padrão *Data Mapper*.
- Permite que o desenvolvedor trabalhe com o modelo de objetos, deixando para a JPA a tarefa de persistir os mesmos no modelo relacional.

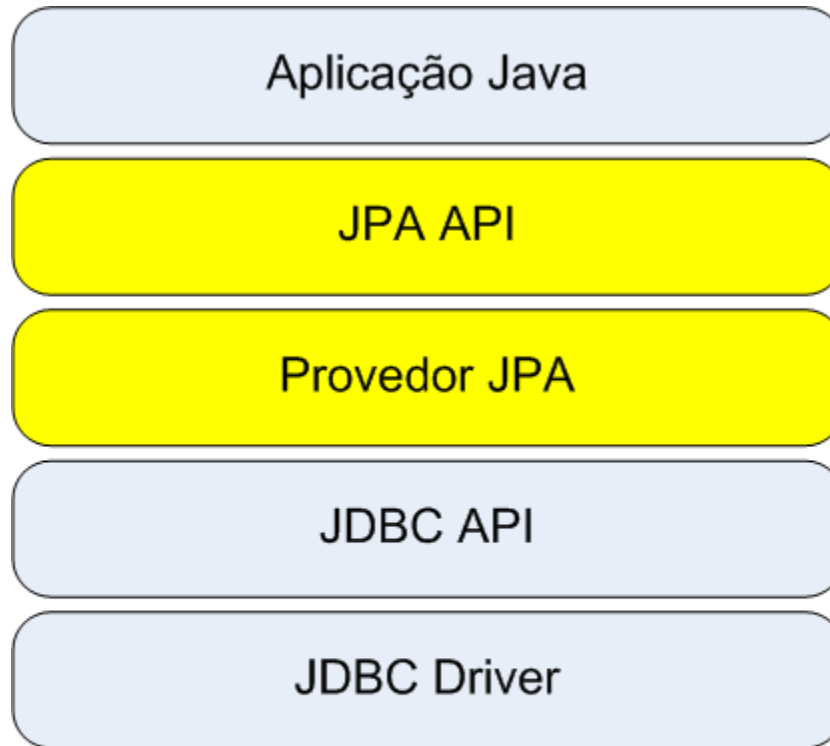
# JPA

- Características principais:
  - Usa **metadados** para orientar o mapeamento entre modelos.
  - Suporta **anotações**.
  - Composta por:
    - Metadados para mapeamento objeto/relacional
    - Persistence API
    - Persistence Criteria API
    - Linguagem de consulta JPQL
- Anotações:
  - Embutidas nos *bytecodes* e lidas em tempo de execução.
  - No caso do JPA são lidas na inicialização do sistema

# JPA

- Implementação concreta, chamada de provedor, é fornecida por diversas fontes:
  - EclipseLink
    - <http://www.eclipse.org/eclipselink/>
  - Oracle Toplink
    - <http://www.oracle.com/technetwork/middleware/toplink/overview/index.html>
  -  • Hibernate ORM
    - <http://www.hibernate.org/>
  - Apache OpenJPA
    - <http://openjpa.apache.org/>

# JPA



# JPA - Conceitos Básicos

- Objetos persistentes são chamados de **entidades**
  - Qualquer classe em Java (conhecidas como POJO – *Plain Old Java Objects*) pode definir um objeto persistente
    - Classe deve possuir pelo menos um construtor sem argumentos, e não pode ser marcada como final
    - Podem explorar herança e classes abstratas
    - Atributos ou propriedades (métodos **get** e **set**) podem ser persistentes (desde que sejam de um conjunto de tipos suportados na JPA)

# JPA - Conceitos Básicos

- Mapeamento objeto-relacional especificado através de anotações
  - As anotações estabelecem a correspondência entre a classe persistente e sua tabela no banco de dados relacional

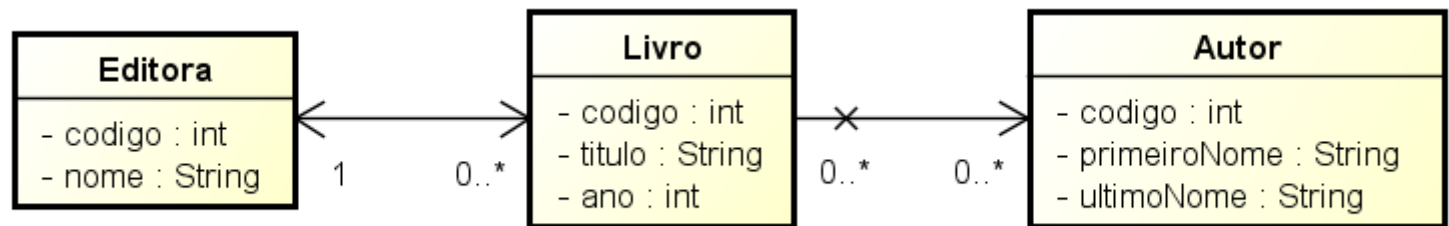


# JPA - Conceitos Básicos

- Anotação *@Entity*:
  - Indica uma entidade persistente
  - Aplicada a uma classe
  - Valor padrão para o nome da tabela é o nome da classe
- Anotação *@Table*:
  - Permite modificar o nome da tabela associada à entidade
  - Atributo *name*

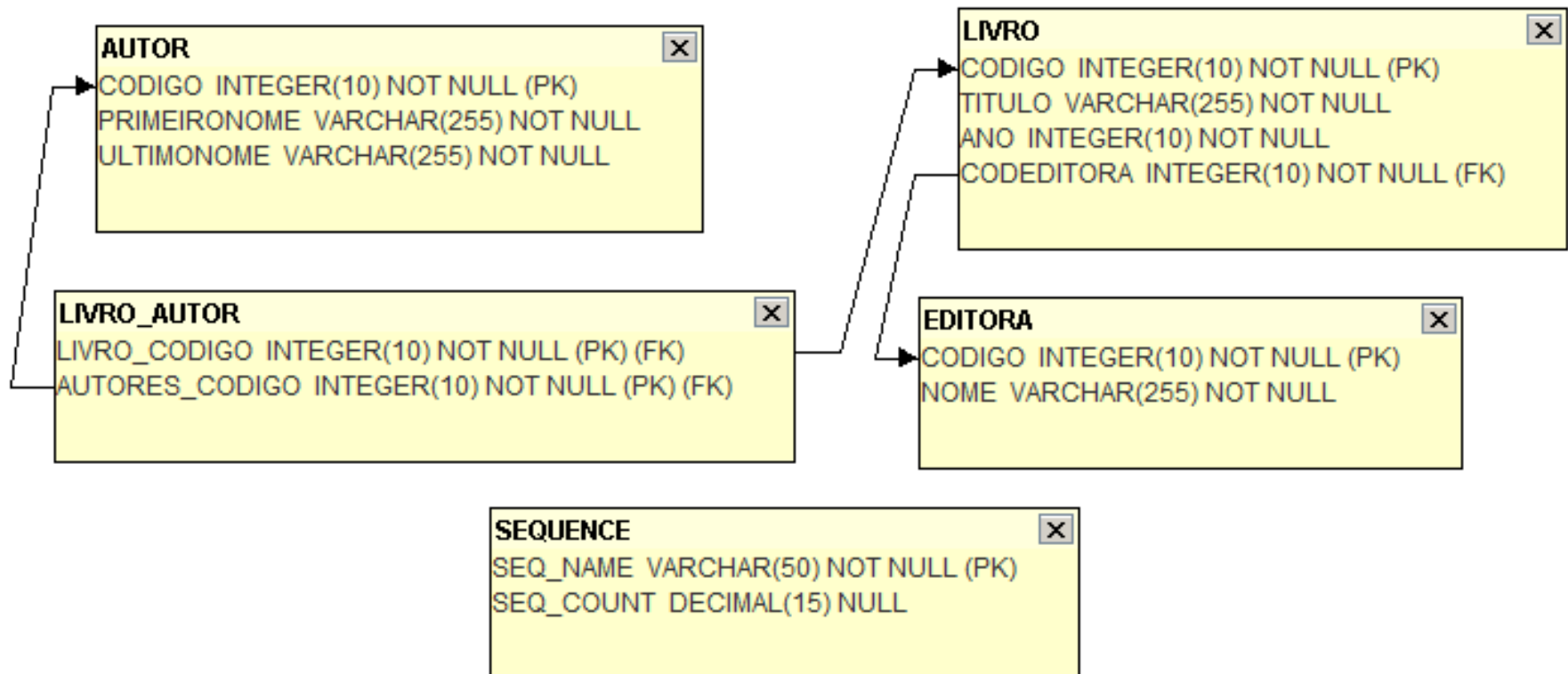
# Exemplo - Livros

- Diagrama de classes de entidades



# Exemplo - Livros

- Diagrama de tabelas geradas automaticamente pelo mapeamento configurado via JPA



# JPA - Conceitos Básicos

- **Ex.:**

```
@Entity  
public class Autor {  
...  
}
```

```
@Entity  
@Table(name="Autores")  
public class Autor {  
...  
}
```

# JPA - Conceitos Básicos

- Mapeamento de atributos:
  - Todos os campos não anotados com *@Transient* (ou de tipo *transient*) são persistentes por padrão
  - Valor padrão para o nome da coluna é o nome do atributo
- Mapeamento de propriedades:
  - Métodos devem seguir o padrão de componentes JavaBeans
  - Mapeamento é aplicado sobre os métodos **get**
  - Valor padrão para o nome da coluna é o nome da propriedade

# JPA - Conceitos Básicos

- Anotação `@Id`:
  - Indica o mapeamento para a chave primária de uma tabela
  - Toda entidade deve possuir uma chave primária
    - Simples – utiliza a anotação `@Id`
    - Composta – utiliza as anotações `@EmbeddedId` e `@IdClass`

# JPA - Conceitos Básicos

- Anotação *@GeneratedValue*:
  - Configura a geração automática de valores para o identificador no momento que novos objetos são persistidos
  - Atributo *strategy* indica o mecanismo de geração a ser utilizado dentre os valores da enumeração *GenerationType*
    - *TABLE* – utiliza uma tabela para geração de identificadores; solução mais portátil pois não depende de mecanismo adicional do banco de dados; anotação *@TableGenerator* para customizar o mapeamento
    - *SEQUENCE* – utiliza um objeto de sequência do banco de dados (se for suportado); anotação *@SequenceGenerator* para customizar o mapeamento
    - *IDENTITY* – utiliza colunas de identidade/auto-incremento no banco de dados (se for suportado)
    - *UUID* – utiliza o tipo de dados UUID de Java

# JPA - Conceitos Básicos

- **Ex.:**

```
@Entity
public class Editora {
    @Id
    private int codigo;
    ...
}
```

```
@Entity
public class Livro {
    @Id
    @GeneratedValue
    private int codigo;
    ...
}
```



# JPA - Conceitos Básicos

- Anotação *@Column*:
  - Modificar propriedades como nome e restrições de integridade
  - Atributo *name* permite modificar o nome da coluna associada ao atributo (ou propriedade)
  - Atributo *nullable* permite configurar a coluna como obrigatória (valor *false*)
  - Atributo *unique* permite configurar a coluna sem valores repetidos
- Anotação *@Basic*
  - É a anotação padrão para informar que o atributo/propriedade é mapeado
  - Modificar propriedades do mapeamento, como *lazy/eager* loading

# JPA - Conceitos Básicos

- **Ex.:**

```
@Entity
public class Editora {
    @Id
    private int codigo;
    @Column(nullable=false)
    private String nome;
    ...
}
```

# JPA - Conceitos Básicos

- Anotação *@Transient*:
  - Indica que o atributo (ou propriedade) não deve ser mapeado para o banco de dados
- Anotação *@Temporal*:
  - Indica que o atributo (ou propriedade) é do tipo associado a valores temporais (*Date* ou *Calendar*) e portanto deve ser mapeado de forma completa (data e hora) ou de forma parcial (somente data ou somente hora)
    - *TemporalType.DATE* – apenas a data (dia, mês e ano)
    - *TemporalType.TIME* – apenas o horário (hora, minutos e segundos)
    - *TemporalType.TIMESTAMP* – a data e o horário (modo padrão)
- Anotação *@Lob*:
  - Utilizada para dados grandes, como fluxos binários de arquivos
  - Aplica-se usualmente a tipos como `byte[]`, `java.sql.Blob`

# JPA - Chaves Compostas

- JPA suporta o conceito de chaves primárias compostas
  - *@IdClass*
  - *@EmbeddedId*

# JPA - Chaves Compostas

- **@IdClass:**
  - Define uma classe separada que possui exatamente os mesmos dados (e nomes) da chave primária composta da entidade
    - Cuidado especial com a implementação dos métodos *equals()* e *hashCode()*
  - Cada elemento da chave primária composta na entidade continua sendo anotado com *@Id*
  - Ex.:

```
public class EmployeePK implements Serializable{  
    private long employeeId;  
    private long companyId;  
    ...  
}
```

```
@Entity  
@IdClass(EmployeePK.class)  
public class Employee {  
    @Id  
    private long employeeId  
    @Id  
    private long companyId  
    ...  
}
```

# JPA - Chaves Compostas

- @EmbeddedId:
  - Define uma classe embutida separada que possui os dados da chave primária composta
    - Cuidado especial com a implementação dos métodos *equals()* e *hashCode()*
  - Ex.:

`@Embeddable`

```
public class EmployeePK implements Serializable{  
    private long employeeId;  
    private long companyId;  
    ...  
}
```

`@Entity`

```
public class Employee {  
    @EmbeddedId  
    private EmployeePK id  
    ...  
}
```

# JPA - Relacionamentos

- Para os diversos tipos de cardinalidades nos relacionamentos entre os objetos tem-se diferentes anotações:
  - 1-1: @OneToOne
  - 1-N: @OneToMany
  - N-1: @ManyToOne
  - N-N: @ManyToMany

# JPA - Relacionamentos

- Além da cardinalidade, o direcionamento do relacionamento traz influência nas configurações das anotações e no comportamento de operações de atualização sobre a base de dados:
  - **Relacionamento unidirecional** – somente uma das entidades envolvidas no relacionamento possui a anotação
  - **Relacionamento bidirecional** – as duas entidades envolvidas no relacionamento devem possuir as anotações



# JPA - Relacionamentos

- Para relacionamento bidirecionais:
  - Primeiro deve-se identificar qual entidade é responsável pelo relacionamento (em inglês utiliza-se o termo “owning-side”)
    - Isso evita a criação de chaves-estrangeiras indevidas
  - O lado inverso da relação deve referenciar o atributo (ou propriedade) do objeto responsável pelo relacionamento através do elemento *mappedBy*
    - Nos relacionamento 1-1 o lado responsável é aquele que possui a chave estrangeira
    - Relacionamentos N-1 não utilizam esse elemento, pois o lado N é sempre o responsável pelo relacionamento
    - Para N-N qualquer lado pode ser o responsável

# JPA - Relacionamentos

- @OneToOne:
  - Relacionamento 1-1
  - Entidade que possui a marcação em um relacionamento implica em uma chave estrangeira na tabela
  - Por padrão o relacionamento não é obrigatório
    - Utilizar `@OneToOne(optional=false)` para implementar cardinalidade mínima 1
  - Anotação `@JoinColumn(name="")` permite alterar o nome da chave estrangeira

# JPA - Relacionamentos

- Ex.: unidirecional 1-1 entre Livro e OfertaEspecial

```
@Entity
```

```
public class Livro {
```

```
    ...
```

```
}
```

```
@Entity
```

```
public class OfertaEspecial {
```

```
    @OneToOne
```

```
    @JoinColumn(name="codigolivro")
```

```
    private Livro livro;
```

```
    ...
```

```
}
```

# JPA - Relacionamentos

- @OneToMany:
  - Relacionamento 1-N
  - Entidade referenciada implica na utilização de uma chave estrangeira
    - Usualmente utiliza-se o mapeamento bidirecional
      - Atributo *mappedBy* indica a referência invertida
      - Cuidado! É tarefa da aplicação manter o relacionamento bidirecional em sincronia!
    - Pode ser utilizado relacionamento unidirecional
      - Anotação *@JoinColumn* para indicar as colunas envolvidas
    - Pode ser utilizada uma tabela de junção (semelhante a N-N)
      - Anotação *@JoinTable* para indicar tabela e colunas envolvidas

# JPA - Relacionamentos

- @ManyToOne:
  - Relacionamento N-1
    - Inverso do @OneToMany em um relacionamento bidirecional
  - Anotação `@JoinColumn(name="")` permite alterar o nome da chave estrangeira

# JPA - Relacionamentos

- Ex.: bidirecional 1-N entre Editora e Livro

```
@Entity
```

```
public class Editora {  
    @OneToMany(mappedBy="editora")  
    private Collection<Livro> livros;  
    ...  
}
```

```
@Entity
```

```
public class Livro {  
    @ManyToOne()  
    private Editora editora;  
    ...  
}
```

# JPA - Relacionamentos

- Ex.: unidirecional 1-N entre Editora e Livro

@Entity

```
public class Editora {  
    @OneToMany  
    @JoinColumn(name="codeditora",  
        referencedColumnName="codigo")  
    private Collection<Livro> livros;  
    ...  
}
```

@Entity

```
public class Livro {  
    @Id  
    private int codigo;  
    ...  
}
```

# JPA - Relacionamentos

- Ex.: unidirecional 1-N entre Editora e Livro com tabela de junção

```
@Entity
```

```
public class Editora {
```

```
    @OneToMany
```

```
    @JoinTable(name="editoralivros",
```

```
        joinColumns=@JoinColumn(name="codeditora"),
```

```
        inverseJoinColumn=@JoinColumn(name="codlivro",  
        unique="true"))
```

```
    private Collection<Livro> livros;
```

```
        ...
```

```
}
```

```
@Entity
```

```
public class Livro {
```

```
    @Id
```

```
    private int codigo;
```

```
    ...
```

```
}
```



# JPA - Relacionamentos

- @ManyToMany:
  - Relacionamento N-N
  - Envolve a utilização de uma tabela de junção
  - Anotação `@JoinTable(name="")` é utilizada para alterar o nome da tabela de junção
    - Atributo `joinColumns` para alterar o nome da chave estrangeira para a origem do relacionamento
    - Atributo `inverseJoinColumns` para alterar o nome da chave estrangeira para o destino do relacionamento

# JPA - Relacionamentos

- Ex.: unidirecional N-N entre Livro e Autor

```
@Entity
```

```
public class Livro {
```

```
    @ManyToMany
```

```
    private Collection<Autor> autores;
```

```
    ...
```

```
}
```

```
@Entity
```

```
public class Autor {
```

```
    ...
```

```
}
```

# JPA - Relacionamentos

- Configurações adicionais:
  - Nos relacionamentos pode ser necessário especificar alguma dependência sobre a outra entidade
    - Utiliza-se o elemento *cascade* cujos valores estão na enumeração *CascadeType* (*PERSIST*, *DETACH*, *MERGE*, *REFRESH*, *REMOVE*, *ALL*)
    - Esse elemento permite configurar o que acontece com as entidades quando uma determinada entidade participante de um relacionamento é alterada, removida ou adicionada
      - O caso mais usual é a remoção em cascata em relacionamentos 1-N, ou seja, quando a entidade de cardinalidade 1 é removida, todas as entidades relacionadas também são
  - É possível especificar múltiplos valores
    - Ex.: *cascade={CascadeType.PERSIST, CascadeType.REMOVE}*

# JPA - Relacionamentos

- Configurações adicionais: ESSENCIAL!
  - Nos relacionamentos pode ser necessário especificar o comportamento do carregamento das entidades relacionadas
    - Ideia básica é o padrão *Proxy* e o padrão *Lazy Load*
    - Deseja-se especificar quando um relacionamento é processado e as entidades associadas carregadas em memória
    - Utiliza-se o atributo *fetch* cujos valores estão na enumeração *FetchType*
      - *Eager* – carrega as entidades associadas para a memória junto a entidade principal do relacionamento (padrão para 1-1, N-1)
      - *Lazy* – carrega as entidades associadas para a memória somente quando o relacionamento for acessado (padrão para 1-N, N-N)
  - Observação: esse atributo pode ser aplicado também a propriedades que não sejam relacionamentos; nesse caso usa-se *@Basic*

# JPA - Relacionamentos

- Ex.: bidirecional 1-N entre Editora e Livro

@Entity

```
public class Editora {  
    @OneToMany(mappedBy="editora",  
        cascade=CascadeType.ALL)  
    private Collection<Livro> livros;  
    ...  
}
```

@Entity

```
public class Livro {  
    @ManyToOne()  
    private Editora editora;  
    ...  
}
```

# JPA - Conceitos Adicionais

- Classes embutidas:
  - Implementam o conceito de composição com cardinalidade 1
  - São utilizadas para representar o estado de uma outra entidade sem a necessidade de serem entidades persistentes em tabelas separadas
    - Por exemplo, uma classe Cep em uma entidade Endereco
      - Os dados de Cep são colunas na tabela Endereco
  - Classe embutida recebe notação *@Embeddable*
  - Atributo de uma entidade do tipo de classe embutida recebe anotação *@Embedded*

# JPA - Conceitos Adicionais

- **Ex.:**

```
@Embeddable  
public class Cep {...}
```

```
@Entity  
public class Endereco {  
    @Embedded  
    private Cep cep;  
    ...  
}
```

# JPA - Conceitos Adicionais

- Atributos de Coleção:
  - Implementam o conceito de composição com cardinalidade N
  - Atributos/propriedades persistentes de uma entidade podem ser representados por coleções (usualmente genéricas) *Collection*, *Set*, *List*, *Map*
  - Atributo/propriedade pode ser anotado com *@ElementCollection*
    - Opcional no caso de uso de coleções genéricas
    - É possível definir o tempo de carga (*fetch*) dos valores como LAZY (valor padrão) ou EAGER
  - *@CollectionTable* e *@Column* podem ser utilizados para mudar o nome padrão do mapeamento para a tabela e colunas utilizadas



# JPA - Conceitos Adicionais

- **Ex.:**

```
@Entity
public class Pessoa {
    private String nome;
    @ElementCollection(fetch=EAGER)
    private Set<String> apelidos;
    ...
}
```

# JPA - Conceitos Adicionais

- **Ex.:**

```
@Entity
public class Pessoa {
    private String nome;
    @ElementCollection
    @CollectionTable(name="ApelidosPessoa",
        joinColumns=@JoinColumn(name="idPessoa"))
    @Column(name="apelido")
    private Set<String> apelidos;
    ...
}
```

# JPA - Conceitos Adicionais

- Enumerações:
  - Por padrão, os valores enumerados são mapeados para valores inteiros (0,1,...) em função da ordem
  - Para que seja utilizado o nome do valor como string, marcar o relacionamento com `@Enumerated(EnumType.STRING)`

# JPA - Conceitos Adicionais

- **Ex.:**

```
@Enum
```

```
public enum Estado {  
    ABERTO, FECHADO, CANCELADO  
}
```

```
@Entity
```

```
public class Pedido {  
    @Enumerated(EnumType.STRING)  
    private Estado status;  
    ...  
}
```

# JPA - Herança

- Entidades podem herdar de classes que não são entidades
- Classes que não são entidades podem herdar de entidades
- Entidades podem ser classes concretas ou abstratas

# JPA - Herança

- JPA utiliza várias estratégias de mapeamento:
  - `InheritanceType.SINGLE_TABLE`
    - Tabela única por hierarquia de classe
  - `InheritanceType.JOINED`
    - Junção de múltiplas tabelas, onde campos/propriedades específicas de uma subclasse são mapeadas para tabelas diferentes daquela utilizada para campos/propriedades da superclasse
  - `InheritanceType.TABLE_PER_CLASS`
    - Uma tabela para cada classe concreta da hierarquia, sem chaves estrangeiras para indicar vínculo entre as classes
- A estratégia deve ser configurada via elemento *strategy* da marcação `@Inheritance` na classe raiz da hierarquia
  - Valor padrão é `SINGLE_TABLE`

# JPA - Herança

- SINGLE\_TABLE:
  - Todas as classes são mapeadas para uma única tabela
  - Cuidado: implica no estado persistente ser *nullable* para atributos das subclasses
  - Tabela deve conter uma coluna (chamada “discriminator column”) com a capacidade de discriminar o tipo concreto da entidade persistida
    - Nome padrão da coluna é DTYPE com o nome da subclasse
    - Coluna especificada via *@DiscriminatorColumn* na classe raiz da hierarquia
    - Elementos configuráveis:
      - name – string para o nome da coluna; nome padrão é DTYPE
      - discriminatorType – tipo da coluna; valores da enumeração *DiscriminatorType* (STRING, CHAR, INTEGER); tipo padrão é STRING
      - columnDefinition – código SQL a ser utilizado na criação da coluna; código padrão é gerado automaticamente pelo provedor
      - length – o tamanho da coluna para valor do tipo STRING; tamanho padrão é 31
  - Valores possíveis especificados via *@DiscriminatorValue* em cada classe de entidade na hierarquia; valor padrão é o nome da classe para tipos STRING

# JPA - Herança

- Ex.: herança por tabela única

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@Table(name = "EMP")
@DiscriminatorColumn(name = "EMP_TYPE")
public abstract class Employee {
    @Id
    private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name = "S_DATE")
    private Date startDate;
    ...
}
```

```
@Entity
@Table(name = "FT_EMP")
@DiscriminatorValue("FTEmp")
public class FullTimeEmployee extends Employee {
    private long salary;
    private long pension;
    ...
}
```



# JPA - Herança

- JOINED:
  - A raiz da hierarquia é representada por uma única tabela com seus campos/propriedades que serão herdados
  - Cada subclasse é mapeada para uma tabela específica que contém somente os campos/propriedades específicos da subclasse
  - Cada tabela de subclasse possui como chave primária uma chave estrangeira que referencia a chave primária da tabela raiz da hierarquia
  - Implica na utilização de joins ao manipular objetos das subclasses
  - Cuidado: alguns provedores utilizam “discriminator column” de forma semelhante à estratégia SINGLE\_TABLE

# JPA - Herança

- Ex.: herança por junção de múltiplas tabelas

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name = "EMP")
public abstract class Employee {
    @Id
    private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name = "S_DATE")
    private Date startDate;
    ...
}

@Entity
@Table(name = "FT_EMP")
public class FullTimeEmployee extends Employee {
    private long salary;
    private long pension;
    ...
}
```

# JPA - Herança

- TABLE\_PER\_CLASS:
  - Cada classe concreta da hierarquia é mapeada para uma tabela
  - Todos os campos/propriedades herdados são mapeados para colunas da tabela específica da classe

# JPA - Herança

- Ex.: herança por tabela por classe concreta

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name = "EMP")
public abstract class Employee {
    @Id
    private int id;
    private String name;
    @Temporal(TemporalType.DATE)
    @Column(name = "S_DATE")
    private Date startDate;
    ...
}

@Entity
@Table(name = "FT_EMP")
public class FullTimeEmployee extends Employee {
    private long salary;
    private long pension;
    ...
}
```

# JPA - Herança

- Classes abstratas:

- Uma classe abstrata pode ser anotada com *@Entity*
- Pode participar de consultas como uma outra entidade qualquer
  - A consulta irá operar sobre todas as subclasses concretas

@Entity

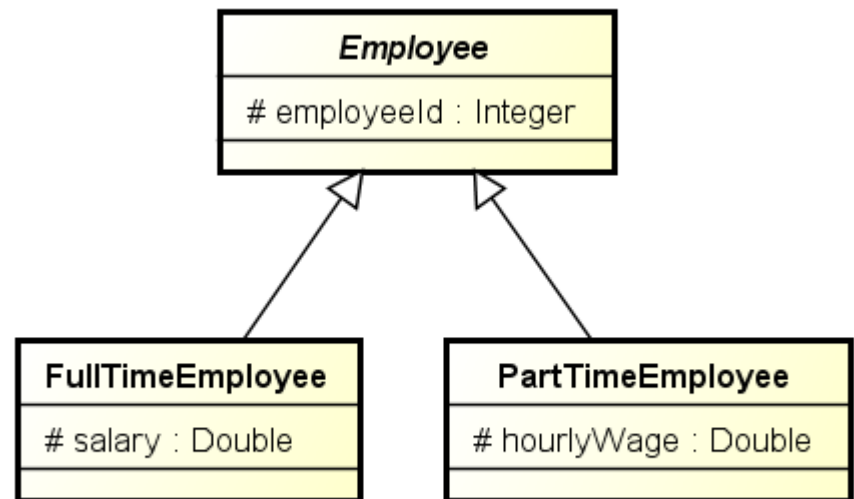
```
public abstract class Employee {  
    @Id  
    protected Integer employeeId;  
    ...  
}
```

@Entity

```
public class FullTimeEmployee extends Employee {  
    protected Double salary;  
    ...  
}
```

@Entity

```
public class PartTimeEmployee extends Employee {  
    protected Double hourlyWage;  
}
```



# JPA - Herança

- Classes que não são entidades:
  - É permitida a herança a partir de classes que não são entidades, mas possuem mapeamento para o estado persistente
  - A classe deve ser anotada com `@MappedSuperclass`

`@MappedSuperclass`

```
public abstract class Employee {  
    @Id  
    protected Integer employeeId;  
    ...  
}  
@Entity  
public class FullTimeEmployee extends Employee {  
    protected Double salary;  
    ...  
}  
@Entity  
public class PartTimeEmployee extends Employee {  
    protected Double hourlyWage;  
}
```

