Anotações Java Persistence API

Objetivos

- Apresentar as anotações que mapeam colunas nas tabelas de um banco de dados;
- Aprender as facilidades oferecidas pelas Enumerações;
- Entender as diferentes estratégias para definição de identificadores únicos (chaves primárias);

Mapeamento de Classes

De uma forma geral, cada classe persistente pode dar origem a uma tabela, onde cada atributo da classe poderá corresponde a uma coluna e os valores dos atributos para cada objeto na classe corresponderemm a uma linha.

No entanto, deve-se, neste mapeamento, ter o cuidado de criar a coluna de identificação, ou seja, a chave primária de cada linha ou registro armazenado. Em um programa Java, quando ocorre a criação de um novo objeto, é gerado automaticamente o identificador deste objeto. Já em um banco de dados relacional cabe ao desenvolvedor a responsabilidade desta criação. Quando ocorre o mapeamento, pode-se armazenar no banco de dados relacional o identificador do objeto como chave primária ou qualquer outro atributo do objeto que possa identificá-lo de forma única.

A forma mais simples de se mapear propriedades de classes é o mapeamento de um simples atributo físico para uma coluna na tabela física. Esta técnica é facilitada através do conhecimento dos tipos básicos de um atributo e de uma coluna. Por exemplo, uma **`String`** pode ter uma correspondência a um tipo **`VARCHAR`** no banco de dados, um tipo **`Date`** OO tem seu correspondente **`Date`** SQL-92 e um tipo **`Short`** OO pode ser mapeado para um tipo **`SMALLINT`** relacional. A correspondência dos tipos é obtida através de quadros de equivalência elaborados pela equipe de arquitetura e padrões do sistema ou fornecidos por fabricantes de ferramentas de mapeamento objeto relacional. Ferramentas MOR, como o Hibernate, fornecem tabelas de correspondência de tipos OO (no caso tipos de dados Java) para tipos de dados SQL.

Nomes de atributos e colunas não necessariamente devem ser iguais, bem como nomes de classes e tabelas. Nem todas as propriedades de uma classe devem ser persistidas em banco de dados. Por exemplo, atributos virtuais, obtidos através de cálculos ou processamento não necessitam ter sua correspondência em colunas.

Entidades

Um entity bean (bean de entidade) normalmente representa uma entidade, seja ela do banco de dados, ou do mundo real. Abaixo mostraremos como criar um bean de entidade para representar um Cliente. No banco de dados haverá uma tabela para armazenar os clientes. A estrutura do bean ficaria assim:

```
package com.targettrust.model;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.*;


@Entity
@Table( name="CLIENTES", uniqueConstraints=@UniqueConstraint(columnNames={"cpf"}))
// nome da entity que corresponde a tabela no banco.
public class Cliente implements Serializable {

	private static final long serialVersionUID = 1L;

	@Id
	// Define a chave primaria
	@GeneratedValue(strategy = GenerationType.AUTO)
	/*
	 * Define uma estratégia para atribuir um valor exclusivo para seus campos
	 * de id automaticamente. Os tipos de estratégias disponíveis são IDENTITY,
	 * SEQUENCE, TABLE, and AUTO. A estratégia padrão é AUTO, cuja aplicação é
	 * deixada ao fornecedor do JPA para implementar.
	 */
	/*
	 * @Version: Define um campo de versão em uma entidade. JPA utiliza um campo
	 * versão para detectar modificações simultâneas para um dados armazenam o
	 * registro. Quando o tempo de execução do JPA detecta múltiplas tentativas
	 * de modificar simultaneamente o mesmo registro, ele lança uma exceção ao
	 * tentar confirmar a última transação. Isso evita que você Sobrescrever a
	 * confirmação anterior com dados desatualizados.
	 */
	private int codigo;

	/*
	 * Define os da coluna no banco
	 * nullable: Se a coluna de banco de dados é anulável.
	 */
	@Column(name = "CPF", nullable = false)
	private long cpf;

	@Column(name = "PRIMEIRO_NOME", nullable = false, length = 50)
	private String primeiroNome;

	@Column(name = "ULTIMO_NOME", length = 50)
	private String ultimoNome;
```

```
    // Quando não definimos @Column é o próprio atributo o nome
    private String rua;

    @Column(name = "NUMERO", nullable = false)
    private String numero;

    private String cidade;

    @Column(name = "CEP", nullable = false)
    private String cep;

    @Column(name = "DATA_ATUALIZACAO")
    @Temporal(TemporalType.DATE)
    private Date dataAtualizacao;

    public Cliente() {      }

    ...geters and seters...
}
```

@javax.persistence.Entity

Você deve ter observado no código acima que esta anotação foi introduzida antes da definição da classe. Ela é responsável por marcar a classe como uma entidade e fazer o mapeamento da mesma para uma tabela do banco de dados.

A tabela que temos no banco de dados para corresponder a esta entidade possui a seguinte estrutura (banco PostgresSql):

```
CREATE TABLE cliente
(
  id_pessoa bigint NOT NULL,
  animal_id_animal bigint,
  codigo integer NOT NULL,
  cep character varying(255) NOT NULL,
  cidade character varying(255),
  cpf bigint NOT NULL,
  data_atualizacao date,
  numero character varying(255) NOT NULL,
  primeiro_nome character varying(50) NOT NULL,
  rua character varying(255),
  ultimo_nome character varying(50),
  CONSTRAINT cliente_pkey PRIMARY KEY (id_pessoa),
  CONSTRAINT fk_1amf7x7t67ds5190bv1aqco1v FOREIGN KEY (id_pessoa)
      REFERENCES pessoa (id_pessoa) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_t693ltg94vqcc2gp84o6ndncj FOREIGN KEY (animal_id_animal)
      REFERENCES animal (id_animal) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT uk_6a8wbkclc7xrljqx6kfcwp108 UNIQUE (cpf)
)
WITH (
  OIDS=FALSE
);
```

Configuração persistence.xml

No padrão JPA temos que configurar o arquivo persistence.xml, com as propriedades para realizar o mapeamento objeto relacional. Neste qrquivos estaram os dados de conexão com o banco, bem como as propriedades do JPA puro ou do Hibernate.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
<persistence-unit name="jpa">
 <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
 <class>com.targettrust.model.Cliente</class>
 <properties>
        <property name="hibernate.connection.driver_class"
       value="org.postgresql.Driver" > </property>
        <property name="hibernate.connection.url"
       value="jdbc:postgresql://localhost:5432/petshop"></property>
        <property name="hibernate.connection.username" value="postgres"
       ></property>
        <property name="hibernate.connection.password" value="postgres"
       ></property>
        <property name="hibernate.dialect"
       value="org.hibernate.dialect.PostgreSQL9Dialect"></property>
        <property name="hibernate.show_sql" value="true"></property>
        <property name="hibernate.format_sql" value="true"></property>
        <property name="hibernate.use_sql_comments" value="false"></property>
        <property name="hibernate.jdbc.wrap_result_sets"
       value="false"></property>
        <property name="hibernate.hibernate.cache.use_query_cache"
       value="true"></property>
        <property name="hibernate.hbm2ddl.auto" value="update"></property>
      </properties>
 </persistence-unit>
</persistence>
```

No Hibernate também existe a possibilidade de configuração em um arquivo chamado: hibernate.cfg.xml

hibernate.hbm2ddl.auto	Valida ou exporta esquema DDL ao banco de dados validate | update | create | create-drop
hibernate.dialect	Define o dialeto de "conversa" no MOR.

Definição de Tabelas

```
@Entity
@Table( name="CLIENTES", uniqueConstraints=@UniqueConstraint(columnNames={"cpf"}))
public class Cliente implements Serializable { }
```

Esta anotação de entidade para uma classe java fará com que a mesma seja mapeada para uma tabela na base de dados que tenha o mesmo nome da classe. Observe que o nome da tabela que temos no banco de dados é CLIENTES e não cliente. Adiante mostraremos como alterar o mapeamento para outro nome de tabela.

```
package javax.persistence;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Entity {
      String name() default "";
}
```

name

Define o nome da classe/entidade. Este nome será utilizado em queries. Por default, o valor que este atributo irá assumir é o nome da classe.

@javax.persistence.Table

Esta anotação foi utilizada em conjunto com a anotação @Entity (a que representa entidades). Quando a anotação @Entity for utilizada sem a anotação `@javax.persistence.Table`, a entidade será mapeada para uma tabela com o mesmo nome da entidade. Caso desejamos mapear a entidade para uma tabela com nome diferente, precisamos utilizar a anotação `@javax.persistence.Table`.

```
package javax.persistence;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Table {
      String name() default "";
      String catalog() default "";
      String schema() default "";
      UniqueConstraint []uniqueConstraints() default {};
}
```

Esta anotação tem como atributos importantes:

name

Define o nome da tabela correspondente no banco de dados. Observe que no exemplo anterior foi utilizado: `@Table(name="CLIENTES")`.

uniqueConstraints

Atributo utilizado para expressar na entidade as constraints que serão adicionadas à DDL gerada para a criação da tabela quando a estratégia de geração de tabela estiver ativa. Veja abaixo um exemplo:

```
import javax.persistence.*;

@Entity
@Table( name="CLIENTES", uniqueConstraints=@UniqueConstraint(columnNames={"cpf"}))
// nome da entity que corresponde a tabela no banco.
public class Cliente implements Serializable {
        ...

}
```

Mapeamento de Propriedades para Colunas

@javax.persistence.Column

A anotação @Column é utilizada para mapear propriedades de uma entidade através de colunas de uma tabela de banco de dados. Por default, o mecanismo de persistência irá procurar por colunas na tabela que tenham o mesmo nome dos atributos da classe/entidade. Em alguns casos, isto pode ser aplicado, porém para sistemas onde as tabelas já existem, definidas por outra pessoa (normalmente tabelas de sistemas legados), o nome das colunas não é um bom nome (explicativo) para os atributos de uma entidade. A interface Column é definida da seguinte maneira:

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Column {
        String name() default "";
        boolean unique() default false;
        boolean nullable() default true;
        boolean insertable() default true;
        boolean updatable() default true;
        String columnDefinition() default "";
        String table() default "";
        int length() default 255;
        int precision() default 0;  // precisão decimal
        int scale() default 0;      // escala decimal
}
```

No exemplo abaixo você pode conferir o uso desta anotação:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="ANIMAIS")
public class Animal {

        @Column( name="APELIDOS", nullable=false, length=100, updatable=true  )
        private String apelido;
        @Column( name="PESO", nullable=false, length=5, updatable=true  )
        private Float peso;
        @Column( name="ALTURA", nullable=false, length=5, updatable=true  )
        private Integer altura;
        @Column( name="STATUS_VIVO", nullable=false, length=1, updatable=true  )
        private boolean vivo;

    ... geters and seters...
}
```

Name:	Define o nome da Coluna.
Nullable:	Especifica se a coluna permite valores nulos.
Insertable:	Especifica se a coluna está sempre incluído em instruções SQL INSERT.
Updatable:	Especifica se esta coluna é sempre incluído nas instruções UPDATE SQL.
Unique:	Define a restrição única para a coluna.
Length:	Define o comprimento da coluna.
Precision:	Define a precisão para os valores da coluna.

O mapeamento irá respeitar os nomes das colunas especificados. Os tipos serão mapeados para os tipos do banco de dados alvo.

Propriedades não Persistentes (Transientes)

@javax.persistence.Transient

Nem sempre você irá querer persistir os dados de todos os atributos de um entity bean. Isto pode ser feito utilizando a anotação `@javax.persistence.Transient` antes da especificação do atributo. Veja abaixo o código da entidade abaixo:

```
import javax.persistence.Table;
import javax.persistence.Transient;
@Entity
@Table(name="ANIMAIS")
public class Animal {

	@Column( name="APELIDOS", nullable=false, length=100, updatable=true  )
	private String apelido;
	@Column( name="PESO", nullable=false, length=5, updatable=true  )
	private Float peso;
	@Column( name="ALTURA", nullable=false, length=5, updatable=true  )
	private Integer altura;
	@Column( name="STATUS_VIVO", nullable=false, length=1, updatable=true  )
	private boolean vivo;
	//Calculo do índice de massa corporal do animal
	@Transient
	private Integer imc;
	...geter and seters...
}
```

Observe que o código acima está utilizando a anotação para indicar ao EntityManager que não persista o atributo. Abaixo, o código para persistência da entidade Animal:

```
public class TestaTransient {

	public static void main(String[] args) {
		EntityManagerFactory factory = Persistence.createEntityManagerFactory("jpa");
		EntityManager fabrica = factory.createEntityManager();

		Animal aninal = new Animal("Rex", 145, 50, true, Especie.Mamíferos);
		fabrica.getTransaction().begin();
		fabrica.persist(aninal);
		fabrica.getTransaction().commit();

		Animal a = fabrica.find(Animal.class,1L);
		System.out.println(a.getImc());

		}

	}
```

Dado no Banco

	id_animal [PK] bigseri	altura integer	apelidos character v	especie character v	peso real	status_vivo boolean
1	1	50	Rex	Mamífero:	145	TRUE
2	2	50	Rex	Mamífero:	145	TRUE
*						

Atributo Identificador (Chave Primária)

@javax.persistence.Id

Essa anotação é utilizada para marcar um atributo como chave primária. `@javax.persistence.Id` deve ser utilizado em combinação com a anotação `@javax.persistence.GeneratedValue`, a qual é utilizada para especificar o esquema gerador de chaves para os objetos/entidades criadas a serem persistidas.

O valor default para a coluna identificadora é "ID", porém este pode ser alterado com a anotação `@javax.persistence.Column`. Muitas vezes, esta alteração será feita.

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Id {
}
```

Observe que esta anotação pode ser adicionada antes do nome do atributo ou antes do método get para o atributo.

```
.....
@Entity
@Table(name="ANIMAIS")
public class Animal {
    @Id
    @GeneratedValue
    @Column(name="ID_ANIMAL")
    private long id;
.............
```

Geração de Chaves Primárias

A geração de chaves para representar de forma única uma entidade na base de dados pode ser feita utilizando as seguintes estratégias:

1) Geração com sequence
2) Geração da chave através de objetos java e uma tabela de apoio

Iremos mostrar abaixo as duas maneiras de geras estas chaves. Observe que alguns bancos podem não oferecer a opção de sequence, desta forma somente a segunda alternativa poderia ser utilizada.

Ambas as formas de geração irão fazer uso de uma anotação `@javax.persistence.GeneratedValue`. Esta será utilizada, de acordo com a estratégia adotada, de forma combinada com outras duas: `@javax.persistence.TableGenerator` e `@javax.persistence.SequenceGenerator`.

@javax.persistence.GeneratedValue

Como já comentamos acima, esta anotação é utilizada em combinação com a `@javax.persistence.Id` para indicar o esquema de geração de identificadores para novos objetos criados. Abaixo o código da interface:

```
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface GeneratedValue {
  GenerationType strategy() default AUTO;
  String generator() default "";
}
```

Observe que esta anotação também pode ser utilizada antes do atributo ou método. Os possíveis atributos são:

strategy

Define o tipo da estratégia de geração do valor. O default é AUTO.

```
import javax.persistence.*;
.......
@Entity
@Table(name="ANIMAIS")
public class Animal {
        @Id
        @GeneratedValue
        @Column(name="ID_ANIMAL")
        private long id;
.............
```

Generator

Define quem será o gerador. Os possíveis valores são:

- IDENTITY: o banco de dados possui um mecanismo/tipos de chaves auto-incrementais, serial (por exemplo: postgreSQL) ou identity.
- SEQUENCE: irá gerar uma SEQUENCE para produzir a chave.
- TABLE: utilizará a anotação @TableGenerator para produzir a chave. Isto exigirá do desenvolvedor a criação de uma tabela. Veremos mais adiante.
- AUTO: será escolhido o esquema de geração com base no banco de dados. Os valores serão os seguintes para cada banco de dados:
  - MySQL - IDENTITY como auto-incremento
  - Postgres - SEQUENCE
  - Oracle - SEQUENCE

Veja abaixo um exemplo para geração automática de chaves. Quando a estratégia de sequence for utilizada, deve ser combinado o uso da anotação `@javax.persistence.GeneratedValue` com a anotação `@javax.persistence.SequenceGenerator`.

```
@Entity
@Table(name = "CHAVES_AUTO")
public class ChavePrimariaAUTO implements Serializable {
        private static final long serialVersionUID = 1L;
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "ID_CHAVE3")
        private int codigo;
        private String nome;
      gets and sets...
}
```

@javax.persistence.SequenceGenerator

Esta anotação irá criar um gerador de chaves com base na sequence informada. Observe que a sequence precisa estar criada no banco de dados, caso contrário será gerada uma exception na hora do deploy da aplicação para indicar que não há sequence. Também, uma boa opção para alguns bancos, como Oracle, que possui um excelente mecanismo de geração de chaves através de sequence.

A interface que representa esta anotação é definida da seguinte maneira:

```
package javax.persistence;

@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)

public @interface SequenceGenerator {
    String name();
    String sequenceName() default "";
    int initialValue() default 0;
    int allocationSize() default 50;

}
```

A propriedade initialValue informa o valor inicial da seqüência. A propriedade allocationsize informa o incremento da seqüência. É preciso informar o gerador da seqüência e linká-lo na anotação SequenceGenerator.

Abaixo um código de exemplo:

```
@Entity
@Table(name = "CHAVES_SEQUENCE")
public class ChavePrimariaSEQUENCE  implements Serializable{

        private static final long serialVersionUID = 1L;

        @Id
        @SequenceGenerator(sequenceName = "CHAVES_SEQUENCE_SEQ",
                name = "sequence")
        @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "sequence" )
        @Column(name = "ID_CHAVE")
        private int codigo;
        gets e sets..
}
```

@javax.persistence.TableGenerator

Para aqueles bancos de dados que não suportam sequence ou mesmo para aqueles desenvolvedores que não são familiarizados ou não gostam de sequence há outra opção. Você pode criar uma tabela na base de dados para armazenar valores de chaves. Abaixo apresentamos a estrutura de uma tabela para isto:

```
CREATE TABLE generator_table (
    PRIMARY_KEY_COLUMN  varchar      NOT NULL,
    VALUE_COLUMN        bigint       NOT NULL
)
```

Observe que esta tabela possui duas colunas, uma para o nome do conjunto de chaves e outra para o valor da chave. O código java na entidade deve fazer uso da anotação `@javax.persistence.TableGenerator` em conjunto com a `@javax.persistence.GeneratedValue`.

Veja a definição completa da anotação:

```
package javax.persistence;

@Target({PACKAGE, TYPE, METHOD, FIELD})
@Retention(RUNTIME)

public @interface TableGenerator {

        String name();
        String table() default "";
        String catalog() default "";
        String schema() default "";
        String pkColumnName() default "";
        String valueColumnName() default "";
        String pkColumnValue() default "";
        int initialValue() default 0;
        int allocationSize() default 50;
        UniqueConstraint[] uniqueConstraints() default {};
}
```

Abaixo um código de exemplo:

```
@Entity
  @Table(name = "CHAVES_TABLE")
  public class ChavePrimariaTABLE implements Serializable{
          private static final long serialVersionUID = 1L;
          @Id
          @TableGenerator(name = "tabela",
               table="GENERATOR_TABLE_CHAVES ",
               pkColumnName="KEY",
               valueColumnName="VALUE",
               pkColumnValue="ID_CHAVE",
               allocationSize=10)
          @GeneratedValue(strategy = GenerationType.TABLE, generator = "tabela")
          @Column(name = "ID_CHAVE")
          private int codigo;


gets e sets ...
}
```

Chaves Compostas

Em algumas situações a chave primária de uma tabela é composta por dois ou mais valores. Nas tabelas deste curso para guardar dados das entidades, temos a tabela MATERIAL_DIDATICO, na qual a chave é composta por código do curso e versão do material (COD_CURSO e VERSAO). A API de persistência provê duas maneiras para isto. Uma é representada pela anotação `@javax.persistence.IdClass` e a outra por `@javax.persistence.EmbeddedId`. Vamos ver em seguida estas duas estratégias.

Nesse capítulo estaremos utilizando a estrutura da seguinte tabela:

```
        cod_curso           varchar(10)  NOT NULL,
        versao              integer      NOT NULL,
        cod_autor           integer,
        dt_prevista_entrega date,
        dt_entrega          date,
        perc_royalty        decimal,
        APOSTILA            bytea,
        APRESENTACAO        bytea,
        SETUP               bytea,
        SOLUCOES            bytea,
        INSTRUCOES          bytea,
        status              varchar(1),
        PRIMARY KEY (cod_curso, versao),
        FOREIGN KEY (cod_autor)             REFERENCES instrutores
(matricula)
```

@javax.persistence.IdClass

Esta é a primeira maneira que temos para definir classes que representem chaves compostas. A classe que representa o bean não ira utilizar esta classe para representar a chave internamente, mas sim para interagir com o gerenciador de persistência (Entity Manager). A estrutura da interface @javax.persistence.IdClass é descrita abaixo:

```
   @Target({TYPE})
   @Retention(RUNTIME)
   public @interface IdClass {
       Class value();
   }
```

Na classe da entidade será designado um ou mais atributos para representarem à chave e estes serão anotados com @javax.persistence.Id. Estas propriedades/atributos precisam ser exatamente do mesmo nome das descritas na classe anotada com @javax.persistence.IdClass. Vejamos agora como podemos criar a classe para representar a chave composta de uma entidade material didático.

```
public class MaterialDidaticoPKIdClass implements Serializable {
      private static final long serialVersionUID = 1L;
      private String codigoCurso;
      private int versao;
      public MaterialDidaticoPKIdClass() {}
      public MaterialDidaticoPKIdClass(int versao, String codigoCurso) {
            this.versao = versao;
            this.codigoCurso = codigoCurso;
      }
      public String getCodigoCurso() {...}
      public void setCodigoCurso(String codigoCurso) {...}
      public int getVersao() {return this.versao;}
      public void setVersao(int versao) {this.versao = versao;}
      public int hashCode() { ... }
      public boolean equals(Object object) { ... }
}
```

A classe para poder representar uma chave composta precisa atender aos seguintes requisitos:

1. Implementar java.io.Serializable
2. Possuir o construtor default
3. Implementar os métodos equals() e hashCode()

Agora vamos ver como fica a classe da entidade que irá anotar as chaves e utilizar esta classe acima apresentada:

```
@Entity
@Table(name = "MATERIAIS_DIDATICOS_IDCLASS")
@IdClass (MaterialDidaticoPKIdClass.class)
public class MaterialDidaticoIdClass implements Serializable {
      private static final long serialVersionUID = 5L;

      @Id
      @Column(name="COD_CURSO")
      private String codigoCurso;
      @Id
      @Column(name="VERSAO")
       private int versao;
      public MaterialDidaticoIdClass() {
      }

      gets and sets ....
}
```

@javax.persistence.EmbeddedId

Outra maneira de se criar uma chave composta é através do processo de embutir dentro da classe bean a classe que define/representa a chave composta. Esta estratégia será a utilizada aqui no curso de JPA.

O código da classe para representar a PK ficaria assim:

```
@Embeddable
public class MaterialDidaticoPKEmbeddable implements Serializable {
        private static final long serialVersionUID = 1L;
        @Column(name = "COD_CURSO", nullable = false)
        private String codigoCurso;
        @Column(name = "VERSAO", nullable = false)
        private int versao;
        public MaterialDidaticoPKEmbeddable() {
        }
        public MaterialDidaticoPKEmbeddable(int versao, String codigoCurso) {
        this.versao = versao;
        this.codigoCurso = codigoCurso;
        }
        Gets and sets...
}
```

A classe da entidade irá declarar um atributo do tipo da classe que representa a PK. Para este "atributo pk" devem ser criados os métodos get e set. Abaixo o código mostrando como ficaria a entidade.

```
@Entity
@Table(name = "MATERIAIS_DIDATICOS")
public class MaterialDidatico implements Serializable {
     private static final long serialVersionUID = 5L;
     @EmbeddedId
     Protected MaterialDidaticoPKEmbeddable pk = new
     MaterialDidaticoPKEmbeddable();
     public MaterialDidaticoPKEmbeddable getPk() {
          return this.pk;
     }
     public void setPk(MaterialDidaticoPKEmbeddable pk) {
          this.pk = pk;
     }
}
```

Enumerações

@javax.persistence.Enumerated

A anotação @Enumerated mapeia tipos enum do Java para o banco de dados.

```
package javax.persistence;
public enum EnumType {
  ORDINAL,
  STRING
}

public @interface Enumerated {
  EnumType value() default ORDINAL;
}
```

Uma propriedade enum pode ser mapeada para a representação string ou para o número ordinal do valor enum. Por exemplo, se nós queremos uma propriedade na entidade Cliente que designe os tipos possíveis de clientes na aplicação, isto poderia ser representado com um enum chamado TipoCliente, com os seguintes valores: PessoaFisica e PessoaJuridica.

```
@Entity
@Table( name="CLIENTES")
public class Cliente implements Serializable {
      ....
      @Column(name = "TIPO_CLIENTE")
      @Enumerated(EnumType.STRING)
      private TipoCliente tipoCliente;;
      ...
}
```

Você pode omitir a anotação @Enumerated do código, mas, neste caso, seria assumido o valor EnumType ORDINAL (default).

Campos de Data e Hora

@javax.persistence.Temporal

A anotação @Temporal fornece informações adicionais ao gerenciador de persistência sobre o mapeamento de uma propriedade `java.util.Date` ou `java.util.Calendar`. Esta anotação permite que você mapeie estes tipos de objetos para um campo `date`, `time` ou `timestamp` no banco de dados. O default para um tipo temporal é o `timestamp`:

```
package javax.persistence;
public enum TemporalType {
   DATE,
   TIME,
   TIMESTAMP
}

public @interface Temporal {
   TemporalType value() default TIMESTAMP;
}
```

Por exemplo, se nós queremos adicionar uma propriedade horaCriação em uma entidade Cliente, deveria ser feito assim:

```
@Entity
@Table( name="CLIENTES")
public class Cliente implements Serializable {
      ....
      @Column(name = "DATA_ATUALIZACAO")
      @Temporal(TemporalType.DATE)
      private Date dataAtualizacao;
      ...
}
```

A propriedade horaCriacao será armazenada na base de dados como um tipo TIME SQL.

Inicializações em Buscas

@javax.persistence.Basic e FetchType

Estas anotações podem ser utilizadas para informar ao gerenciador de persistência que não deve ser retornado no instante de criação/recuperação da entidade todos os dados daquela entidade, ou seja, alguns dados podem ser retornados no momento em que formos acessar o método get, por exemplo.

FetchType é uma enumeration que contém os valores LAZY e EAGER. Uma propriedade anotada como sendo LAZY só receberá o valor do banco de dados quando a mesma for acessada. Já se for anotada como EAGER, os dados serão setados na hora da busca da entidade. Veja abaixo a interface:

O LAZY indica que o relacionamento/atributo não será carregado do banco de dados de modo natural. Ao fazer busca na entidade, seus relacionamentos ou atributos marcados, não serão carregados. Desse modo a quantidade de retorno de dados vinda do banco de dados será menor. A vantagem dessa abordagem é que a query fica mais leve, e o tráfego de dados pela rede fica menor.

```
package javax.persistence;

@Target({Method, FIELD})
@Retention(RUNTIME)
public @interface Basic {
  FetchType fetch() default EAGER;
  boolean optional() default true;
}
```

Observe que a interface possui um atributo optional(). Pode ser interessante usar este atributo quando o container irá gerar o banco de dados, uma vez que se posto optional como true, a coluna poderá ser nula.

Veja agora um exemplo do uso de LAZY:

```
@Entity
@Table( name="CLIENTES")
public class Cliente implements Serializable {@OneToOne

        @Basic(fetch = FetchType.LAZY)
        private Animal animal;
}
```