**Orientação a Objetos**

**Composição**

Na composição codificamos pequenos comportamentos padrão e usamos composição para definir comportamentos mais complexos. Com isso podemos alterar a associação entre classes em tempo de execução, assim um objeto pode assumir mais de um comportamento.

Quando, em uma linguagem de programação, os objetos são tipados, os tipos geralmente podem ser divididos em tipos compostos e não-compostos e a composição pode ser considerada uma relação entre os tipos: um objeto de um tipo composto (e.g. *carro*) "tem um" objeto de um tipo mais simples (e.g. roda).

**Formas de identificar de quando usar composição:**

...TEM UM(A)...

* Carro TEM UM( motor.
* Carro TÊM portas.
* Casa TEM UMA uma cozinha.

**Pilares:**

**Encapsulamento:**

Conceitua-se **encapsulamento** como sendo o processo utilizado para proteger os campos e operações de uma classe (atributos e métodos), permitindo que apenas os membros públicos - em **Java** métodos Get / Set - sejam acessados pelos usuários de determinada classe.

Encapsulamento é o ato de privar(deixar oculto) algo que só pode ser acessado pelo próprio objeto, mas não pode ser acessado por outro objeto que faz referencia ao objeto que tem uma classe privada.

**Modificadores de Acesso:**

**Public:**

Ao criar um atributo publico, significa que qualquer classe do sistema pode acessar o atributo ou método publico de outra classe.

**Protected :**

Um atributo protegido é visível no próprio pacote, na própria classe e é transmitido por herança para um outro pacote, ou seja, ele só é visível em outro pacote desde que haja herança .

**Package(Default):**

Um atributo de nível pacote(padrão), só pode ser acessado por outra classe do mesmo pacote, e não por outras classes de pacotes diferentes.

**Private:**

Ao criar um atributo privado dentro da classe, ele só pode ser acessado dentro da classe, ou seja, não pode ser acessado por outras classes.

**Herança:**

Herança é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos, através de "heranças". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.

Na Herança uma classe pode herdar os atributos e métodos de sua superclasse. A herança é transitiva, então uma classe pode herdar de outra classe que herda de outra classe, e assim por diante, até uma classe base. Subclasses podem substituir alguns métodos e/ou atributos para alterar o comportamento padrão.

**Formas de identificar de quando usar Herança:**

...É UM(A)...

* Civic É UM carro.
* Gato É UM mamífero.
* Maça É UMA fruta.

Super Classe (+Genérica)

Sub Classe (+Específica)

**Polimorfismo:**

**Polimorfismo** é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.

Polimorfismo Estático:

Polimorfismo Estático = Sobrecarga.

Um polimorfismo estático seria quando você escreve estaticamente o mesmo método de diferentes formas diferentes(cada um com sua assinatura diferente) por exemplo.

Poliformismo Dinâmico:

Para ter um polimorfismo dinâmico, precisa ter Herança.

Exemplo:

Tipo especifico:

Civic c = new Civic();

Tipo mais genérico:

Carro c = new Civic();

c = new Ferrari();

Ao criar uma variável mais genérico, você pode atribuir outros tipos diferente para o objeto, uma hora ele pode ser um Civic, outra hora pode ser uma Ferrari, por o tipo dele ser Carro, e Ferrari e Civic serem herança em Carro.

Mas não pode criar uma variável do tipo Civic e atribuir para ela uma Ferrari.

**Abstração:**

A **abstração** é o ponto de partida para a criação de programas utilizando **POO**. Trata-se da capacidade de extrair dos personagens ou dos itens presentes no contexto, suas principais características, criando, dessa forma, objetos.

Se estivéssemos lidando com um software que precisará armazenar dados de um carro, poderíamos pensar nos seguintes métodos:

* Frear.
* Acelerar.
* Buzinar.
* Acender Farol.

Além de métodos, este nosso carro também teria propriedades, que poderiam ser

* Cor
* Quantidade de Portas
* Combustível utilizado
* Modelo
* Marca

A abstração é, portanto, a captura das principais características do personagem ou item envolvido no contexto, de forma que elas possam ser facilmente descritas em uma classe, que gerará um objeto.

**ENUM**

São tipos de campos que consistem em um conjunto fixo de constantes (static final), sendo como uma lista de valores pré-definidos. Na linguagem de programação **Java**, pode ser definido um tipo de enumeração usando a palavra chave **enum**. Todos os tipos **enums** implicitamente estendem a classe **java**.

Os enums começam com o valor 0 por padrão, e o próximo enum tem o valor 1 por padrão, e assim por diante na ordem em que eles estão, mas também é possível atribuir um valor para os enums, deixando seu valor fixo, independente da ordem em que estejam.

public enum Direcao {

NORTE,

LESTE,

SUL,

OESTE

}

Por padrão as enums são escritas todas em maiúsculas, e para iniciar os valores declarados dentro das variáveis Enum, é preciso [**declarar um construtor**](https://www.devmedia.com.br/construtores-em-java-primeiros-passos/28618) para iniciar os seus atributos que são declarados:

public enum Direcao {

NORTE(11),

LESTE(12),

SUL(13),

OESTE(14);

Direcao(int valor) {

}

}

**Interface**

Em java uma “Interface” sempre vai ter métodos públicos e abstratos, ou seja, mesmo que você não diga explicitamente que o método da interface é publica, ela é publica implicitamente, e é abstrata pois não tem corpo.

Exemplo:

public interface Esportivo {

void ligarTurbo();

void desligarTurbo();

}

public interface Ferrari implements Esportivo {

}

Sendo possível uma classe herdar de mais de uma interface.

public interface Ferrari implements Esportivo, Luxo {

}

Sendo obrigatório as classes que herdam de uma interface implementarem seus métodos que não foram implementados, da mesma forma que uma classe que herda de outra classe abstrata ela é obrigada a implementar o método que não foi finalizada do classe pai.

Para definir um método “padrão” dentro de um interface, se usa a palavra reservada “default”:

public interface Luxo {

void ligarAr();

void desligarAr();

default int velocidadeDoAr() {

return 1;

}

}

**Classes abstratas**

Pode-se dizer que as [**classes abstratas**](http://www.devmedia.com.br/classe-abstrata-curso-basico-de-java-e-orientacao-a-objetos-parte-29/25322) servem como “modelo” para outras classes que dela herdem, não podendo ser instanciada por si só. Para ter um objeto de uma classe abstrata é necessário criar uma classe mais especializada herdando dela e então instanciar essa nova classe. Os métodos da classe abstrata devem então serem sobrescritos nas classes filhas.

Por exemplo, é definido que a classe “Animal” seja herdada pelas subclasses “Gato”, “Cachorro”, “Cavalo”, mas ela mesma nunca pode ser instanciada.

**Lambdas**

Uma expressão **Lambda** permite definir uma interface funcional (novamente, um método abstrato) que o compilador identifica pela estrutura. O compilador pode determinar a interface funcional representada a partir de sua posição. O tipo de uma expressão **lambda** é o da interface funcional associada.

**Interface funcional:**

I**nterfaces funcionais** são **interfaces** que têm um método a ser implementado, em outras palavras, um método abstrato. Isso significa que toda **interface** criada que respeite esta premissa, tornando-se automaticamente uma **interface funcional**. E para ser uma interface funcional, ela tem que ter apenas 1 único método abstrato dentro dela.

**ForEach:**

Para usar uma função lambda com um forEach é bem prático, em vez de fazer um forEach tradicional como:

List<String> aprovados = Arrays.asList(“Ana”, “Bia”, “Lia”, “Gui”):

for ( String nome : aprovados) {

System.out.println(nome);

}

Basta colocar da seguinte forma:

aprovados.forEach(nome -> System.out.println(nome));

Dessa forma você tem um forEach no formato de função lambda, sendo muito mais prático e rápido de se usar, fazendo a mesma coisa de primeiro forEach.

Também tem o “Method Reference” ou “Método por Referencia”, é quando vc passa dentro do forEach uma referencia para o valor, em vez de passar explicitamente o parâmetro:

aprovados.forEach(System.out::println);

Dessa forma ele vai fazer a mesma coisa que os outros 2 forEach, mas de forma por referencia.

**Stream API**

Entre as diversas funcionalidades adicionadas à linguagem **Java** em sua versão 8 está a **Streams API**, recurso que oferece ao desenvolvedor a possibilidade de trabalhar com conjuntos de elementos de forma mais simples e com um número menor de linhas de código.Isso se tornou possível graças à incorporação do [**paradigma funcional**](http://www.devmedia.com.br/programacao-funcional-com-java/32176), combinado com as [**expressões lambda**](http://www.devmedia.com.br/trabalhando-com-expressoes-lambdas-no-java-8/34322), o que facilita a manutenção do código e aumenta a eficiência no processamento devido ao uso de paralelismo.

A [**proposta em torno da Streams API**](http://www.devmedia.com.br/streams-api-trabalhando-com-colecoes-de-forma-flexivel-em-java/31980) é reduzir a preocupação do desenvolvedor com a forma de implementar controle de fluxo ao lidar com coleções, deixando isso a cargo da API. A ideia é iterar sobre essas coleções de objetos e, a cada elemento, realizar alguma ação, seja ela de filtragem, mapeamento, transformação, etc.

Exemplo:

List<String> aprovados = Arrays.asList(“Lu”, “Gui”, “Luca”, “Ana”);

Stream<String> stream = aprovados.stream();

stream.forEach(System.out::println);

Aqui está sendo passado para a variável stream do tipo “Stream”, o retorno do método “stream()” da lista “aprovados”, e depois sendo usado o “forEach” para imprimir cada elemento da stream.

**Map:**

Retorna uma stream consistindo no resultado da aplicação de uma função de mapeamento nos elementos da Stream.

A Stream “Map” vai transformar cada elemento da Stream, ou seja, ele vai gerar uma nova Stream transformada para oque foi passado na lambda expression.

Exemplo:

Consumer<String> print = System.out::print;

List<String> marcas = Arrays.asList(“BMW “, “Audi “, “Honda “);

marcas.stream().map(m -> m.toUpperCase()).forEach(print);

No exemplo, a lista “marcas” será feito um mapeamento e criado uma nova Stream contendo todas as Strings da lista “marcas”, só que tudo em maiúsculo.

**Filter:**

Filtra os elementos de acordo com uma condição retornando uma nova Stream. A Stram “Filter”, recebe uma lambda com uma condição, e retorna todos os elementos da Stream que atendem a condição passada na lambda.

Exemplo:

Aluno a1 = new Aluno(“Ana”, 7.8);

Aluno a2 = new Aluno(“Gui”, 6.8);

Aluno a3 = new Aluno(“Daniel”, 9.8);

Aluno a4 = new Aluno(“Bia”, 5.8);

Aluno a5 = new Aluno(“Pedro”, 8.8);

List<Aluno> alunos = Arrays.asList(a1, a2, a3, a4, a5);

alunos

.stream()

.filter(a -> a.nota >= 7)

.map(a -> “Parabéns “ + a.nome + “! Você foi aprovado(a)!”)

.forEach(System.out::println);

Nesse exemplo o filter retornará uma nova Stream com todos os alunos que tem nota maior ou igual a 7, e depois será transformada em uma String pelo map, e por último será impresso na saída do sistema cada um dos elementos da nova Stream pelo forEach.

**Reduce:**

Realiza uma operação de redução que leva uma sequência de elementos de entrada e os combina em um único resultado, como acumular valores. A Strream Reduce faz uma soma, tendo como parâmetro o “total”, e o valor “atual”, onde o total é o acumulador, e o valor atual é o próximo item da lista que vai ser somado com o valor de total, retornando um único valor.

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

BynaryOperator<Integer> soma = (total, atual) -> total + atual;

Integer total = nums.stream().reduce(soma).get();

System.out.println(total);

O valor retornado será 45, que é a soma de todos os valores da lista “nums”, e como não foi passado o valor inicial, o retorno será um “Optional<Integer>”, sendo necessário usar o “.get()” para pegar o valor retornado.

Integer total2 = nums.parallelStream().reduce(100, soma);

System.out.println(total2);

Nesse outro exemplo, como foi passado o valor inicial, o retorno do reduce já será convertido para o tipo Integer, mas como foi usado o “parallelStream” em vez do “stream”, ele vai retornar “945”, pois o “parallelStream” vai passar para cada próximo elemento da lista um “novo” valor inicial, sendo melhor usado apenas o “stream”.

**Match:**

**allMatch:**

Retorna *true* se todos os elementos da stream correspondem ao predicado fornecido.

Aluno a1 = new Aluno(“Ana”, 7.8);

Aluno a2 = new Aluno(“Gui”, 6.8);

Aluno a3 = new Aluno(“Daniel”, 9.8);

Aluno a4 = new Aluno(“Bia”, 5.8);

Aluno a5 = new Aluno(“Pedro”, 8.8);

List<Aluno> alunos = Arrays.asList(a1, a2, a3, a4, a5);

Predicate<Aluno> aprovados = a -> a.nota >= 7;

System.out.println(alunos.stream().allMatch(aprovados));

Se todos os lementos da lista forem verdadeiros, o allMatch retorna verdadeiro, se não ele retorna falso.

**anyMatch:**

Retorna *true* se pelo menos um dos elementos da stream corresponde ao predicado fornecido.

System.out.println(alunos.stream().anyMatch(aprovados));

Se pelo menos um dos elementos da lista for verdadeiro, o anyMatch retorna verdadeiro, se não ele retorna falso.

**noneMatch:**

 Retorna *true* se nenhum dos elementos da stream corresponde ao predicado fornecido.

Predicate<Aluno> reprovados = aprovados.negate();

System.out.println(alunos.stream().noneMatch(reprovados));

Se todos os elementos da lista não forem verdadeiros, o noneMatch retorna verdadeiro, se não ele retorna falso.

**Tratamento de erros e exceções**

**Exceções são** todos os erros que ocorrem durante o processamento de um método e podem ser esperados ou não esperados. Como o próprio nome já diz "Exceptions" **são** as possíveis **exceções**. **São** falhas que, idealmente, não devem ocorrer no fluxo do sistema.

Para tratar um erro em Java, se coloca o possível erro dentro de um try/catch, ou seja, oque pode provavelmente gerar o erro fica dentro do try, e o retorno do erro fica dentro do catch, como uma mensagem de alerta do erro, ou algo do gênero, exemplo:

try {

System.out.println(7 / 0);

} catch(ArithmeticException e) {

System.out.println(“Ocorreu um erro: “ + e.getMessage()) ;

}

Nesse exemplo, como é impossível dividir um número por zero, será retornado a mensagem de erro mais o erro que foi causado, do tipo ArithmeticException, podendo colocar de forma especifica o tipo do erro, ou de forma genérica(Exception).

Para você lançar uma exceção em um tratamento de erro, se usa a palavra reservada “throw”, exemplo:

static void gerarErro1() {

throw new RuntimeException(“Ocorreu um erro bem legal #01”);

}

Dessa forma, assim que o programa que estiver em execução chegar nesse ponto do código, será exibido a mensagem de erro na saia do sistema, juntamente com a mensagem padrão do Java, e para fazer o tratamento, se usa o try/catch da mesma forma que no primeiro exemplo:

try {

gerarErro1();

} catch(RuntimeException e) {

System.out.println(e.getMessage()) ;

}

Dessa forma, será exibido apenas a mensagem de erro passada dentro da Exception. Sendo “gerarErro1()” uma exceção não checada ou não verificada.

Exceção checada:

static void gerarErro1() throws Exception {

throw new RuntimeException(“Ocorreu um erro bem legal #02”);

}

Quando se tem uma exceção checada, você é obrigado a colocar na assinatura do método que você lança o tipo de exceção, usando a palavra reservada “throws” seguido dos fechamentos do método e o tipo da exceção que poderá ser lançada. Caso você não coloque que o método pode lançar a exceção do tipo definido na frente da palavra “throws”, você deve tratar o erro no mesmo instante envolvendo a exceção dentro de um bloco try/catch:

static void gerarErro1() {

try {

throw new RuntimeException(“Ocorreu um erro bem legal #02”);

} catch(Exception e) {

System.out.println(“Que legal!”) ;

}

}

Checked exceptions(Exceções checadas) são utilizadas para erros recuperáveis enquanto que Unchecked exceptions(Exceções não checadas) são utilizadas para erros irrecuperáveis. Significa dizer que quando você sabe que seu erro pode ser tratado, você utiliza Checked Exceptions, caso contrário utilize Unchecked Exceptions.

**Finally:**

O finally, serve para quando você quer que determinado código seja executado, independente de ter tido erro ou não, como por exemplo fazer o fechamento da conexão com o banco de dados, mesmo que tenha acontecido um erro antes da conexão ser fechada, exemplo:

Scanner entrada = new Scanner(System.in);

try {

System.out.println(7 / entrada.nextInt());

} catch(Exception e) {

System.out.println(e.getMessage());

} finally {

System.out.println(“Finally...”);

entrada.close();

}

System.out.println(“Fim!”);

Nesse exemplo, independente de acontecer o erro ou não, o código que está dentro do finally vai acontecer, como o fechamento do “Scanner” e o “System.out.println”.

**Generics**

**Generic** é uma maneira de criar parâmetros para classes e definir tipos que podem ser substituídos em vários lugares do programa. Isso elimina o uso da classe **Object** para definirmos tipos que podem variar no decorrer do programa.

Por exemplo, vamos imaginar uma classe com um método que possa receber uma **string**, **int** ou **decimal** como parâmetro. A única maneira de criar uma única função deste tipo seria criar uma que recebesse um **Object**, já que o Object pode ser de qualquer tipo. Na hora de recuperarmos o valor, o programa deveria fazer uma conversão correta em tempo de execução (operação chamada **Cast**) e, caso o tipo contido no retorno da função não fosse o correto, certamente teríamos um erro.

Exemplo:

public class Caixa<T> {

private T coisa;

public void guardar(T coisa) {

this.coisa = coisa;  
 }

public T abrir() {

return this.coisa;

}

}

Na classe “Caixa”, se passa o generics dentro do maior e meno(< e >), por uma letra, e todos os lugares da classe que ele será usado.

public class CaixaTeste {

public static void main(String[] args) {

Caixa<String> caixaA = new Caixa<>();

caixaA.guardar("Segredo");

String coisaA = caixaA.abrir();

System.out.println(coisaA);

Caixa<Double> caixaB = new Caixa<>();

caixaB.guardar(3.1415);

Double coisaB = caixaB.abrir();

System.out.println(coisaB);

}

}

Ao instanciar a classe com um generic, se passa o tipo do generics que a classe vai receber, fazendo assim com que substitui o lugar da letra que foi passada na classe generics pelo tipo no momento da instanciação da classe, como se faz al criar uma collection.

Generics com Herança:

public class CaixaInt extends Caixa<Integer> {

}

Nesse exemplo, a classe “CaixaInt” herda todas as funcionalidade da classe “Caixa” passando os generics do tipo “Integer”.

Generics com restrição:

public class CaixaNumero<N extends Number> extends Caixa<N> {

}

Nesse exemplo, foi herdado todas as funcionalidades da classe “Caixa” para a classe “CaixaNumero” recebendo um generics, e passando para a classe “Caixa”, porém com uma restrição, aceitando apenas tipo que herdam da classe “Number”.

Generics em métodos:

public class ListaUtil {

public static <T> T getUltimo(List<T> lista) {

return lista.get(lista.size() - 1);

}

}

Para passar um generics em um método statico, se passa o nome do generics(No caso a letra “T”), o retorno do método que pode ser o própria generics, e no caso desse exemplo, também passa o generics como o tipo da Lista.

public class ListaUtilTeste {

public static void main(String[] args) {

List<String> langs = Arrays.asList("JS", "PHP", "Java", "C++");

List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6);

String ultimaLinguagem = ListaUtil.getUltimo(langs);

System.out.println(ultimaLinguagem);

Integer ultimoNumero = ListaUtil.<Integer>getUltimo(nums);

System.out.println(ultimoNumero);

}

}

Apenas pelo tipo que é passado na declaração da variável no exemplo, já faz a conversão automático, mas pode ser passado o tipo entre a notação ponto e o método no modo de definição do generics, fazendo assim com que deixe “Amarrado” o tipo do valor recebido.

**Modulos**

À medida que os problemas vão se tornando maiores e mais complexos, sempre é possível simplificar dividindo a solução em partes menores, chamadas de subprogramas (em Java, métodos). Cada parte menor do problema tem uma implementação mais simples, favorecendo a legibilidade e a manutenibilidade do subprograma.

No momento que é criado um projeto java, pode ser definido o modulo daquele projeto, pode ser definido o nome do modulo como “app.calculo”, ou após ter criado o projeto sem ter definido o módulo, pode também criar um arquivo com o nome “module-info.java” para definir os módulos a serem exportados ou importados, exemplo:

module app.calculo {

exports br.com.cod3r.app.calculo;

}

Normalmente o modulo tem o mesmo nome do projeto separando por ponto(.), e dentro do “module”, é definido todos os pacotes que serão exportados, como no exemplo acima, será exportado todas as classe, interfaces e enums que estiverem dentro do capote “br.com.cod3r.app.calculo”, mas caso tenha outro sub-pacote, não será exportado por padrão os arquivos desse sub-pacote, dentro que colocar outro “exports” para cada pacote especifico. Já para usar as classes do pacote exportado, no outro projeto, onde tem o arquivo “module-info.java”, deve ser importado cada classe, interface ou enum do pacote que ele está, exemplo:

module app.financeiro {

requires app.calculo;

}

Nesse exemplo, será importado todos os arquivos que estão dentro do pacote “app.calculo”, fazendo assim com que possar ser utilizado corretamente qualquer classe, interface ou enum publicas dentro do outro projeto.

Para exportar não apenas o pacote, mas também todas as dependências de outros módulos daquele pacote, se usa a palavra “transitive”, fazendo assim com que além de exportar o pacote, também exporte a dependência do pacote de outro projeto, exemplo:

module app.calculo {

requires transitive app.logging;

exports br.com.cod3r.app.calculo;

}

Dessa forma, tudo que está sendo importado do projeto “app.logging”, ao importar o projeto “app.calculo”, será importado de forma transitiva também o “app.logging”.

Podendo também exportar um pacote para um módulo especifico, por exemplo:

module app.calculo {

requires transitive app.logging;

exports br.com.cod3r.app.calculo;

exports br.com.cod3r.app.calculo.interno to app.financeiro;

}

Fazendo assim com que apenas o modulo “app.financeiro” do projeto “app-financeiro” consiga utilizar o pacote “interno”.

É possível também deixar o módulo “aberto” para todos os outros módulos que dependem dele, exemplo:

open module app.calculo {

requires transitive app.logging;

exports br.com.cod3r.app.calculo;

exports br.com.cod3r.app.calculo.interno to app.financeiro;

}

Dessa forma, tudo dentro do módulo “app.calculo” estará aberto para poder ser alterados até mesmo atributos privados da classe usando “Field” através de reflexão, mas também é possível deixar aberto apenas um pacote especifico, exemplo:

module app.calculo {

requires transitive app.logging;

exports br.com.cod3r.app.calculo;

exports br.com.cod3r.app.calculo.interno to app.financeiro;

opens br.com.cod3r.app.calculo;

}

Fazendo assim com que todos os módulos que dependem do módulo “app.calculo”, tem o pacote “br.com.cod3r.app.calculo” aberto para eles, mas também é possível deixar aberto para um ou mais módulos específicos, separados por vírgula:

module app.calculo {

requires transitive app.logging;

exports br.com.cod3r.app.calculo;

exports br.com.cod3r.app.calculo.interno to app.financeiro;

opens br.com.cod3r.app.calculo to app.financeiro;

}

Dessa forma, o pacote “br.com.cod3r.app.calculo” estará aberto apenas para o módulo “app.financeiro”.

**Annotations**

Anotações é um recurso usado para anotar classes, campos e métodos, de tal maneira que essas marcações podem ser tratadas pelo compilador, ferramentas de desenvolvimento e bibliotecas. Três anotações, denominadas tipos anotação padrão, são parte do pacote **java.lang**, e podem ser usadas sem qualquer esforço adicional: **@Override**, **@Deprecated** e **@SuppressWarnings**. Além dessas, o texto trata das anotações de anotações – **@Retention**, **@Documented**, **@Target** e **@Inherited** – as quais são também parte da API juntamente com as anteriores, e da personalização de tipos anotação, mostrando como criar nossos próprios tipos.

Todos os exempos abaixo estão dentro do pacote “javax.persistence”.

**@Entity**

Ao marcar uma classe com a annotation **@Entity**, está dizendo que a classe é mapeada para uma entidade do banco de dados.

**@Entity**

public class Produto {}

**@Id**

Ao marcar um atributo(instância) da classe com a annotation **@Id**, está dizendo que o atributo é a chave primaria do banco de dado, exemplo:

**@Id**

Private Long Id;

**@GeneratedValue**

Ao marcar um atributo(instância) da classe com a annotation **@GeneratedValue**, está dizendo que o atributo será auto incremento no banco de dados, usado juntamente da annotation **@Id,** podendo também definir estratégias como:

**@Id**

**@GeneratedValue**(strategy = GenerationType.IDENTITY)

Private Long Id;

Definindo a estratégia como “IDENTITY”, está dizendo que cada entidade(tabela) do banco de dados tenha sua própria sequencia de Id.

**@GeneratedValue**(strategy = GenerationType.SEQUENCE)

Definindo a estratégia como “SEQUENCE”, está dizendo que ele pode e vai compartilhar uma mesma sequência com mais de uma entidade.

Se não definir nenhuma estratégia ele usará por padrão o “GenerationType.AUTO”, que faz com que ele crie a tabela “hibernate\_sequence” para poder fazer o incremento dos IDs.

**@Transient**

Ao marcar um atributo(instância) da classe com a annotation **@Transient**, está dizendo que o atributo não será mapeado para o banco de dado.

**@Table**

Ao marcar uma classe com a annotation **@Table**, está dizendo que a classe é mapeada para uma entidade do banco de dados, podendo colocar um nome especifico da tabela diferente do nome da classe, como:

**@Table**(name = “produtos”)

public class Produto {}

Caso você não coloque a annotation “**@Table**” definindo o nome da tabela a ser mapeada, por padrão a tabela que será criado será com o mesmo nome da classe. Podendo também definir qual schema do banco de dados essa tabela está vinculado, por exemplo:

**@Table**(name = “produtos”, schema = “curso\_java”)

public class Produto {}

Fazendo assim com que fique arramado essa tabela com o schema definido.

**@Column**

Ao marcar um atributo(instância) da classe com a annotation **@Column**, está dizendo que aquele atributo é para ser mapeado para a coluna especificada na annotation, exemplo:

**@Column**(name = “prod\_nome”, length = 200, nullable = false)

private String nome;

O atributo “name” define o nome da coluna a ser mapeada, o “length” define o tamanho máximo da coluna(caso a instância da classe seja uma String, será criado a coluna do tipo VARCHAR com o tamanho especificado), caso não seja especificado o “length” ele usá o valor padrão que é “255”, e o “nullable” é para dizer se a coluna aceita valores NULL ou não, como foi definido “false” significa que ela não aceita valores NULL.

Para atributos do tipo Double, se usa as propriedades da annotation:

**@Column**(name = “prod\_preco”, nullable = false, precision = 11, scale = 2)

private Double preco;

Onde “precision” é o tamanho do campo(a precisão do número), por exemplo: 12345678901, e “scale” é a quantidade de números depois da vírgula, por exemplo: 123456789,01.

**@OneToOne**

Ao marcar um atributo(instância) da classe com a annotation **@OneToOne**, está dizendo que o atributo é um relacionamento “Um para Um”, fazendo assim com que a coluna seja criado como uma chave estrangeira(FOREIGN KEY), sendo automaticamente criado a coluna com o nome do atributo + \_id, exemplo:

**@OneToOne**

private Assento assento;

Dessa forma a coluna criada na tabela terá o nome “assento\_id”. Essa annotation deve ser usada na classe em que será mapeada a tabela que ficará o relacionamento “Um para um”. Para fazer um relacionamento Bidirecional no mundo OO, se usa o atributo “mappedBy” dentro da annotation “@OneToOne” para dizer qual o atributo da outra classe que está mapeando o relacionamento no banco de dados, exemplo?

@OneToOne(mappedBy = “assento”)

private Cliente cliente;

Sendo o valor de “mappedBy” o nome exato do atributo da classe que está fazendo o mapeamento.

**@JoinColumn**

Ao marcar um atributo(instância) da classe com a annotation **@JoinColumn**, você pode definir a o atributo que refere-se a coluna da tabela com a propriedade “unique = true” para dizer que essa coluna não aceita repetição, exemplo:

**@JoinColumn**(name = “assento\_id”, unique = true)

private Assento assento;

Dessa forma a coluna criada na tabela não aceitara valores duplicados.

**@ManyToOne**

Ao marcar um atributo(instância) da classe com a annotation **@ManyToOne**, está dizendo que o atributo é um relacionamento “Muitos para Um”, fazendo assim com que a coluna seja criado como uma chave estrangeira(FOREIGN KEY), sendo automaticamente criado a coluna com o nome do atributo + \_id, exemplo:

**@ManyToOne**

private Pedido pedido;

**@ManyToOne**

private Produto produto;

Dessa forma a coluna criada na tabela terá o nome “pedido\_id” e “produto\_id”. Essa annotation deve ser usada na classe em que será mapeada a tabela que ficará o relacionamento “Muitos para um”.

**@ManyToMany**

Ao marcar um atributo(instância) da classe com a annotation **@ManyToMany**, está dizendo que o atributo é um relacionamento “Muitos para muitos”, exemplo:

**@ManyToMany**

private List<Sobrinho> sobrinhos;

**@ManyToMany**(mappedBy = “sobrinhos”)

private List<Tio> tios;

Dessa forma será criado uma terceira tabela com o nome “tio\_sobrinho” pelo fato de o mapeamento estar na classe “Tio” que tem o atributo “sobrinhos”.

**@JoinTable**

Ao marcar um atributo(instância) da classe com a annotation **@JoinTable**, você pode definir o nome da tabela de junção do relacionamento “muitos para muitos”, ou seja, se usa a annotation “**@JoinTable**” juntamente da notação “**@ManyToMany**”, exemplo:

**@ManyToMany**

**@JoinTable**(name = “atores\_filmes”)

private List<Ator> atores;

Podendo também definir o nome das colunas que serão criadas juntamente com a tabela passando outras propriedades como:

**@ManyToMany**

**@JoinTable**(name = “atores\_filmes”, joinColumns = **@JoinColumn**(name = "filme\_id", referencedColumnName = "id"), inverseJoinColumns = **@JoinColumn**(name = "ator\_id", referencedColumnName = "id"))

private List<Ator> atores;

Onde “joinColumns” é as informações da primeira coluna, como “name” sendo o nome a ser dado a coluna da tabela criado, e “referencedColumnName” é o nome da coluna a ser referenciada, e “inverseJoinColumns” é as informações da segunda coluna tendo as mesmas propriedades de “joinColumns”, definindo o nome da coluna e pegando o nome da coluna da tabela que será referenciada. Ficou nessa ordem pois a classe “Filme” está fazendo o mapeamento para a tabela a ser criada, se fosse feito o mapeamento pela classe “Ator” seria definido ao contrário os valores.

**@Embeddable**

Ao marcar uma classe com a annotation **@Embeddable**, está dizendo que todos as classes que tem um atributo do tipo da classe marcada com a annotation “**@Embeddable**” que está sendo mapeada para uma tabela do banco de dados, será adicionado as informações da classe na tabela da classe que tem uma instância dessa classe, e não criando uma tabela separada, exemplo:

**@Embeddable**

public class Endereco {}

Todas as classes que tiverem um atributo do tipo “Endereco”, como:

private Endereco enderco;

Será criado uma coluna para cada atributo da classe “Endereco” dentro da classe que tem uma instância do tipo “Endereco” dentro da tabela que a classe está mapeando, e não criado uma tabela “Endereco”.

**@Inheritance**

Ao marcar uma classe com a annotation **@Inheritance**, estará definindo como será gerado as tabelas das classes por que contenham herança, por exemplo:

**@Entity**

**@Inheritance**(strategy = InheritanceType.SINGLE\_TABLE)

**@DiscriminatorColumn**(name = “tipo”, length = 2, discriminatorType = DiscriminatorType.STRING)

**@DiscriminatorValue("AL”)**

public class Aluno {}

**@Entity**

**@DiscriminatorValue("AB")**

public class AlunoBolsista extends Aluno {}

Onde “**@Inheritance**” define a estratégia “InheritanceType.SINGLE\_TABLE” criando uma tabela para as duas classes, “**@DiscriminatorColumn**” define as informações da coluna, como o nome da coluna(name), o tamanho dos valores inseridos nela(length) e o tipo da coluna(String que será convertido para VARCHAR), e “**@DiscriminatorValue**” define o valor que será adicionado na coluna “tipo” para objetos do tipo “Aluno”, que seria o tipo mais geral, enquanto na classe “AlunoBolsista” define o “**@DiscriminatorValue**” como “AB” para objetos de um tipo mais especifico. Caso use a estratégia “InheritanceType.JOINED” no “**@Inheritance**”, será criado 2 tabelas, uma tabela “Aluno” e uma “AlunoBolsista”, enquanto a tabela “Aluno” terá todos os atributos da classe “Aluno” como coluna, e a tabela “AlunoBolsista” terá os atributos da classe “AlunoBolsista” fazendo uma chave estrangeira para a coluna “Aluno”, tendo um relacionamento de “um para um”.

**@RestController**

Ao marcar uma classe com a annotation **@RestController**, estará dizendo que a classe é um controlador do tipo Rest, exemplo:

**@RestController**

public class PrimeiroController {}

Fazendo assim com que através dessa classe, seja possível fazer o mapeamento das rotas

**@RequestMapping**

Ao marcar um método da classe com a annotation **@RequestMapping**, estará dizendo que aquele método retornará/exibirá algo em alguma rota definida, caso não defina a rota, será exibido na rota padrão que a “/”.

**@RestController**

public class PrimeiroController {

**@RequestMapping**

public String ola() {

return “Olá Spring Boot!”;

}

}

Dessa forma, sera exibido na rota padrão(/), a mensagem que está sendo retornado “Olá Spring Boot!”, e o método padrão é o método “GET”, podendo também passar em qual rota será retornado valor, e também o método daquela rota, exemplo:

**@RestController**

public class ClienteController {

**@RequestMapping**(path= “/ola”)

public String ola() {

return “Olá Spring Boot!”;

}

}

Dessa forma, será exibido na rota “/ola” o retorno do método “olá”, é possível dizer também qual o método dessa rota, usando:

**@RestController**

public class PrimeiroController {

**@RequestMapping**(method = RequestMethod.GET, path= “/ola”)

public String ola() {

return “Olá Spring Boot!”;

}

}

Dessa forma estará dizendo de forma explicita que o método da requisição é do tipo “GET”. Também é possível usar o “**@RequestMapping**” para mapear a classe inteira, dizendo qual a rota padrão daquela classe, exemplo:

**@RestController**

**@RequestMapping**(path= “/clientes”)

public class ClienteController {

**@GetMapping**(path= “/qualquer”)

public Cliente obterCliente() {

return new Cliente(28, “Pedro”, “123.456.789-00”;

}

}

Dessa forma não é necessário colocar “clientes/qualquer” no “path” do método “obterCliente”, pois todos os métodos por padrão estarão na rota “/clientes/...” e o nome da rota que foi definido em algum método dentro dessa classe, pois a classe já definiu por padrão que a rota é “/clientes”.

**@GetMapping**

Ao marcar um método da classe com a annotation **@GetMapping**, estará dizendo que aquele método retornará/exibirá algo em alguma rota definida sendo o método da requisição do tipo “GET”, caso não defina a rota, será exibido na rota padrão que a “/”, da mesma forma que a annotation “**@RequestMapping**”.

**@RestController**

public class PrimeiroController {

**@GetMapping**

public String ola() {

return “Olá Spring Boot!”;

}

}

Da mesma forma que a annotation “**@RequestMapping**”, sera exibido na rota padrão(/), a mensagem que está sendo retornado “Olá Spring Boot!”, sendo esse método do tipo “GET”, por estar usando a annotation que explicita exatamente o tipo do método como “GET”.

**@RestController**

public class PrimeiroController {

**@GetMapping**(path= “/ola”)

public String ola() {

return “Olá Spring Boot!”;

}

}

Dessa forma a mensagem “Olá Spring Boot!”, não será mais exibida na rota padrão(/), e sim na rota “/ola”. Também é possível passar uma array de argumentos, passando todas as rotas em que aquele método retornará algum valor, exemplo:

**@RestController**

public class PrimeiroController {

**@GetMapping**(path= {“/ola”, “/saudacao”})

public String ola() {

return “Olá Spring Boot!”;

}

}

Dessa forma será retornado a String “Olá Spring Boot!” nas duas rotas.

**@PostMapping**

Ao marcar um método da classe com a annotation **@PostMapping**, estará dizendo que aquele método é uma requisição do tipo “POST” como o próprio nome já diz, fazendo com que ele faça a requisição do tipo “POST”, ou seja, faz um “insert”, exemplo:

**@RestController**

public class PrimeiroController {

**@GetMapping**(path= {“/ola”, “/saudacao”})

public String ola() {

return “Olá Spring Boot!”;

}

**@PostMapping**(path= “/saudacao”)

public String saudacao() {

return “Olá Spring Boot!”;

}

}

Dessa forma ele não exibirá a mensagem “Olá Spring Boot!”, e sim deve fazer algum “POST” nesse método, tanto pelo corpo da página ou pela rota recebendo um parâmetro.

**@PutMapping**

Ao marcar um método da classe com a annotation **@PutMapping**, estará dizendo que aquele método é uma requisição do tipo “PUT” como o próprio nome já diz, fazendo com que ele faça a requisição do tipo “POST”, ou seja, faz um “update”, exemplo:

**@RestController**

**@RequestMapping**( “/metodos”)

public class PrimeiroController {

**@PutMapping**(path= “/saudacao”)

public String put() {

return “Requisição PUT”;

}

}

O método “PUT” é utilizado para fazer uma alteração no servidor, ou seja, alterar algo do banco de dados.

Tipos de Requisições:

**GET** -> Obter Dados

**POST** -> Inserir Dados

**PUT** -> Atualizar Dados de forma total

**PATCH** -> Atualizar Dados de forma parcial

**DELETE** -> Excluir Dados

**OPTIONS** -> Retorna os métodos HTTP que a URL suporta

**TRACE** -> Requisição de teste

**HEAD** -> Parecido com o “**GET**”, recebe a mesma resposta, mas sem o conteúdo, usado para saber se a URL está funcionando.