**PROGRAMACAO ORIENTADA A OBJETOS**

- Antes da POO, o que se usava era a programação estruturada (Linux, Windows pertencem na linguagem C, que faz parte de um paradigma estruturado)

**- PE:**

Tem uma representação mais simplista

Foca em operações e dados – com manipulação de dados brutos

Voltada para o “como fazer”

**POO:**

Mais realista (possui mais mecanismos e mecanismos mais avançados na linguagem)

Foca na modelagem de entidades e interações entre estas

Voltada para “o que fazer”

Vantagens: melhor coesão (unidades de códigos com responsabilidades bem definidas), melhor acoplamento, diminuição do gap semântico (ou seja, os objetos do mundo real ficam mais próximos do mundo computacional no POO), coletor de lixo (em POO há liberação de espaço constantemente de forma automática)

- a POO é um paradigma de analise(entender o que fazer, qual o problema), projeto(pegar o problema e resolver através de um projeto computacional, solucionando o problema) e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.

**FUNDAMENTOS**

1. **ABSTRAÇÃO**: Abstrair é a habilidade de pensarmos nas características mais essenciais e deixar de lado as acidentais, pra pensar no que mais pode ser reusado. “Processo pelo qual se isolam características de um objeto, considerando os que tenham em comum certos grupos de objetos”
2. **REUSO**: é a capacidade de criar novas unidades de código a partir de outras já existentes
3. **ENCAPSULAMENTO**: é capacidade de esconder complexidades e proteger dados – criando camadas p dificultar o acesso aos dados

EXERCÍCIO: Levando em consideração uma loja online de livros, modele uma entidade livro

Primeiro deve-se pensar que a modelagem da entidade vai depender do nicho em que se está trabalhando.

Deve-se pensar em todas as características possíveis q possam envolver a entidade livro:

- entidade LIVRO: qtd de paginas, ano de publicação, edição, peso, tipo de capa, assunto, tipo, gramatura, idioma, tema, editora, autor, coloração, ISBN, sub-titulo, dimensões, qtd de capítulos, acabamento, tipo de papel...

Depois deve-se afunilar e pensar em quais características são de fato essenciais, considerando o contexto/nicho do problema ou solução que se quer ter:

- entidade LIVRO: qtd de paginas, ano de publicação, edição, ~~peso~~, ~~tipo de capa~~, assunto, tipo, ~~gramatura~~, idioma, tema, editora, autor, ~~coloração~~, ISBN, sub-titulo, ~~dimensões~~, qtd de capítulos, ~~acabamento, tipo de papel~~...

Proposta de entidade LIVRO final:

- entidade LIVRO: qtd de paginas, ~~ano de publicação~~, edição, ~~peso~~, ~~tipo de capa~~, assunto, ~~tipo~~, ~~gramatura~~, ~~idioma~~, tema, editora, autor, ~~coloração~~, ISBN, sub-titulo, titulo, ~~dimensões~~, ~~qtd de capítulos~~, ~~acabamento, tipo de papel~~...

**A ESTRUTURA DA POO**

Estruturas básicas da POO: CLASSE, ATRIBUTO, OBJETO, MÉTODO, MENSAGEM

**CLASSE**: É uma estrutura estática, um molde que abstrai um conj de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis desse objeto através de atributos. Ou seja, uma classe descreve os serviços providos por seus objetos e quais informações podem armazenar

Ex: bola / carro / viagem / venda / comprador...

class Carro {

}

Dicas p nomear: substantivos, nomes significativos e dentro do contexto

**ATRIBUTO**: Elemento de uma classe responsável por definir sua estrutura de dados. O conjunto desses será responsável por representar suas características e fará parte dos objetos criados a partir da classe. Ou seja, é através do atributos que conseguimos caracterizar a classe.

Ex: classe: carro , atributo: cor

Dicas p nomear: substantivos, adj, nomes significativos, contexto deve ser considerado, abstração, tipos adequados (entender qual a finalidade do atributo pra poder saber qual o tipo adequado pra ele)

class Carro {

int portas;

}

**MÉTODO**: É uma porção de código disponibilizada pela classe. É executado qdo feita uma requisição a ele. Um método serve p identificar quais serviços e acoes que a classe oferece. São responsáveis por definir e realizar um determinado comportamento.

Cria-se um método apenas quando precisa.

VOID é o tipo de retorno que o método tem, nesse caso VOID NÃO RETORNA NADA.

Dicas p nomear: verbos, nomes significativos e dentro do contexto

A criação do método precisa ter a visibilidade, retorno, nome e parâmetros( caso precise).

A ASSINATURA de um método é o nome + parâmetros

SOBRECARGA: é qdo vc muda o parâmetro do método sem alterar seu nome

class Carro {

void frear( ) {

............

}

}

**2 MÉTODOS ESPECIAIS:**

**Construtor**: serve p criar objetos e guardar parâmetros. em JAVA, o construtor é obrigatório ter o mesmo nome da classe

class Carro {

Carro( ) {

............

}

}

**Destrutor**: Auxilia na destruição do objeto através do coletor de lixo, que antes apagar chama essas classes de objetos p liberar espaço na maquina. Tem que ter a palavra VOID p dizer q não tem retorno + a palavra “finalize” p dizer que quer destruir

class Carro {

void finalize() {

............

}

}

**OBJETO**: É a representaocao do conceito/entidade do mundo real, que pode ser física (bola,carro,arvore) ou conceitual(viagem, compra, venda, estoque) e possui um significado bem definido p um determinado software. p/ esse conceito/entidade deve ser definida inicialmente uma classe a partir da qual posteriormente serão instanciados objetos distintos.

São os objetos que estarão na memoria do computador e fazer as coisas acontecerem. São instâncias da classe

Carro carro = new Carro (); 🡪 AQUI CRIOU-SE UM ATRIBUTO OU VARIÁVEL CHAMADO carro (2) PARA A CLASSE do tipo Carro (1) e atribuiu-se um construtor (new) criando um objeto/instância nova p armazenar naquela variável do começo. Ou seja, estou criando um objeto e armazenando na variável carro através de um construtor default – ou seja, um construtor que tem o mesmo nome da classe e sem parâmetro, por isso n tem nada nos parênteses.

**MENSAGEM**: é o processo de ativação de um método de um objeto.Ocorre quando uma requisição (chamada) a esse método é realizada, assim disparando a execução de seu comportamento descrito por sua classe. Pode ser tb direcionada diretamente à classe, caso a requisição seja um método estático.

Carro carro = new Carro (); 🡪 criei um objeto / é pegar uma classe e instanciar um obj nela

carro.<método>; 🡪 chamei o método que eu quero

Carro.<método>; 🡪 aqui chamei o método diretamente à classe e não ao objeto

**AS RELAÇÕES: HERANÇA, ASSOCIAÇÃO E INTERFACE**

**HERANÇA**: É o relacionamento entre classes em que uma classe chamada de subclasse ( ou classe filha, derivada) é uma extensão, um subtipo de outra classe chamada de superclasse (classe pai, mae, superbase). Devido a isso, a subclasse consegue reaproveitar os atributos e métodos dela. Além dos que venham a ser herdados, a subclasse tb pode definir seus próprios membros.

Se uma coisa for SUBTIPO da outra, a tendência é usar herança. Se uma coisa vai usar a outra, a tendência é usar associação.

class A extends B { 🡪 A está herdando de B, A é uma extensão, um subtipo de B

..........

}

**HERANÇA SIMPLES**: A classe filha só tem uma classe mãe

**HERANÇA MÚLTIPLA**: A classe filha tem uma ou mais classes mães

**UPCAST**: Subir na sua hierarquia de classes até chegar na classe mais genérica/ampla. Mt usado

A a = new B(); 🡪 B é uma subclasse de A. Defini uma variável do tipo a na classe A e transformo B – classe filha em A-classe mae. Ou seja, fiz um Upcast

**DOWNCAST**: Descer na sua hierarquia de classes até chegar na classe mais específica. Pouco usado

B a = (B) new A(); 🡪 B é uma subclasse de A. Defini uma variável do tipo a na classe B e transformo A-classe mae em B-classe filha. Ou seja, fiz um Downcast

**POLIMORFISMO**: A mesma ação/método se comportando de forma diferente.

Ex: tenho uma entidade PAGAMENTO, que contem várias formas de processamento (processar () ) – Pix, Boleto, Cartão de crédito. Ou seja, o mesmo método/ação – no caso processar – se comportando de forma diferente de acordo com a instância/objeto em questão.

**SOBRESCRITA**: A mesma ação podendo se comportar de forma diferente.

@override

**ASSOCIAÇÃO**: Possui um relacionamento entre classes/objetos, no qual esses possam pedir ajuda a outras classes/objetos e representar de forma completa o conceito ao qual se destinam. Nesse tipo de relacionamento as classes interagem entre si p atingir seus objetivos.

**ASSOCIAÇÃO ESTRUTURAL**: tem a ver com os atributos

- **Composição:** “Com parte todo” – a parte só existe com o todo, uma parte é composta com a outra. Um depende do outro p existir

Ex: pessoa e endereço

class Pessoa {

Endereco endereço;

}

-**Agregação**: “Sem parte todo” - uma parte pode existir sem o todo

Ex: Disciplina e Aluno

Class Disciplina {

Aluno aluno;

}

**ASSOCIAÇÃO COMPORTAMENTAL:** tem a ver com uma associação dos métodos

-Dependência: “Depende de” -

**INTERFACE**: Define um contrato que deve ser seguido pela classe que a implementa. Qdo uma classe implementa uma interface, ela se compromete a realizar todos os comportamentos que a interface disponibiliza.

Ex: oq faz uma impressora funcionar em qq sistema operacional? É a INTERFACE, como se fosse um contrato, lá existirão métodos que deverão ser implementados e assim conseguem se comunicar.

Interface A {

......

}

class B implements A {

.......

}

**A ORGANIZAÇÃO DE PACOTES E VISIBILIDADES**

**PACOTES**: Uma organização física/logica criadas para separar classes com responsabilidades distintas. Com isso, espera-se que a aplicação fique mais organizada e seja possível separar classes de finalidades e representatividades diferentes.

Ex: pacotes de classes de negócios, classes de utilitários, classes de regras de negócios, etc..

package <nomedopacote> 🡪 para criar um pacote

import <nomedopacote> 🡪 importar uma classe que está em outro pacote, a tornando visível

**VISIBILIDADES:** Modificadores de acesso. Serve p determinar até que ponto uma classe, atributo ou método pode ser usado.

TIPOS: usar de acordo com a sua necessidade, com a visibilidade adequada

**PRIVATE**: mais restritiva. Só dentro da classe

Private int i;

private void do();

**PROTECTED**: dentro da classe, mesmo pacote e subclasses

protected int i;

protected void do ();

**PUBLIC**: visível em qq lugar, sem restrição

Public int i;

Public void do();