**08/08:**

* Na POO, a atenção principal é com os dados que o programa irá tratar;
* Identificar classes e objetos que serão manipulados;
* Pensar quais serão os atributos (campos) e os comportamentos (métodos);
* Na POO, a atenção principal é com os dados que o programa irá tratar
* Identificar classes e objetos que serão manipulados
* Pensar quais serão os atributos (campos) e os comportamentos (métodos)

**15/08:**

**PILARES:**

* Reutilização: ao invés de repetir várias vezes o mesmo dado, cria-se uma classe e reutiliza a mesma, modelando conforme necessário;
* Encapsulamento: não permite o acesso a dados de qualquer forma. Não é viável ter acesso a modificar o atributo diretamente. Quando o modificador de acesso do atributo é mudado para private, pode-se alterar a varíavel somente em um local específico;
* Especialização: melhora-se aquilo que se tem e inclui-se novos dados;

**VARIÁVEIS DO TIPO PRIMITIVAS:**

* Define o tipo específico da variável que está sendo criada;
* Um método só retorna o valor desejado se trocar o 'void' para o tipo de varíavel desejado. Ou seja, void = sem retorno nenhum, não altera a variável no main.
* Caso 1 - é mostrado na tela o valor 5

public static void Somar(int num) {

num = num + 7;

}

* Caso 2 - é mostrado na tela o valor 12 (5+7)

public static int Somar(int num) {

num = num + 7;

return num;

}

* Caso 3:

public static void main{

int num = 5;

Somar(num);

sysout("o valor da variavel é " + num)

}

1. Reutilização

2. Encapsulamento

3. Especialização

* Int, char, boolean, int num = 5;
* Não possui new;
* Guarda o conteúdo;

**VARIÁVEL DO TIPO REFERÊNCIA:**

* Guarda a posição da memória, não o conteúdo;
* Sempre começa com letra maiúscula (como as classes);
* Possuem new;
* **Exemplo:**
* int vetor[] = new vetor[4];
* umaPessoa = new Pessoa();
* Está atrelado ao new;
* String;

**MÉTODOS:**

* São as funções/comportamentos de uma classe (EX: imprimir, popular);
* Servem para otimizar os códigos, utilizando menos linhas, criando assim, um código mais limpo
* public static void main = PSVM
* Inicia com letra minúscula;
* Assinatura de um método:
* Modificador de acesso + retorno + nome + (lista de parâmetros)
* "Verificar" ou "Persistir" = if
* Sobrecarregar um método é um conceito da programação orientada a objetos onde você pode definir múltiplas versões de um método com o mesmo nome na mesma classe, mas com diferentes parâmetros.
* As variáveis vetor.length e vetor[], não são as mesmas, pois cada método tem um espaço diferente na memória do código, portanto os nomes podem ser diferentes;
* Quando o método é void, NÃO retorna;
* Quando o método não for void, obrigatoriamente retorna;

**CLASSE:**

* É quando se tem mapeamento/referência do mundo real (EX: Pessoa, Endereço);
* Os atributos são características das classes (EX: peso, idade, nome, altura);
* Numa classe, a variável vira um atributo;
* Inicia com letra maiúscula;
* Quando uma classe é criada, ela é capaz de receber diferentes tipos de variáveis (int, string...) e agrupar vários tipos de informações, enquanto uma variável e um vetor pode receber somente um tipo;
* Uma classe é dividida em:
* Nome (pessoa)
* Atributos (características como nome, idade)
* Métodos (comportamentos como andar, set, get)
* A classe se materializa em um objeto onde é possível colocar informações dentro das variáveis
* A vantagem do encapsulamento é fazer tudo somente uma vez, sem precisar repetir x vezes
* Não se printa nada dentro de uma classe
* Tratamento de erro é sempre tratado na interface.
* Na classe fazer tratamento de exceção, e no main deve-se capturar a exceção e mostrar uma mensagem através do try e catch.
* **Ordem:**
* Nome da classe: classe se transforma no objeto;
* Atributos (características): são as variáveis.
* Public: qualquer um pode acessar;
* Private: é um atributo privado, encapsulado, só é possível ter acesso a ele por meio de um atributo (get e set);
* Métodos (comportamentos): é onde se faz o tratamento;
* Usa-se this quando trata-se de um atributo da classe (ex: this.nome = nome);
* Escopo: variáveis;
* Quando não há informação, deixar o campo vazio, pois o zero pode ser uma informação inserida pelo usuário (disparar erro);
* Não se printa nada dentro de uma classe;
* **Exceção:** throw new IllegalArgumentException – não é viável, pois não apresenta o erro;

**CRIANDO UM OBJETO:**

**Métodos get (Accessors):**

* Propósito: Permitir a leitura de um atributo privado de uma classe por código externo;
* Encapsulamento: Através dos métodos get, pode-se controlar como um atributo é acessado. EX: em vez de retornar diretamente um objeto mutável (como uma lista), você pode retornar uma cópia dele para evitar alterações indesejadas;
* Flexibilidade: Se a forma como um atributo é armazenado ou calculado mudar, o método get pode ser atualizado sem afetar o código que o utiliza;

**Métodos set (Mutators):**

* Propósito: Permitir a modificação de um atributo privado de uma classe por código externo;
* Validação: Antes de alterar o valor de um atributo, pode-se usar o método set para verificar se o novo valor é válido. EX: se um atributo "idade" não deve ser negativo, essa verificação pode ser feita no set.
* Eventos: Se algo precisa acontecer quando um atributo é alterado (por exemplo, notificar observadores ou atualizar outros campos relacionados), isso pode ser implementado no método set;
* Encapsulamento: os métodos set garantem que o estado interno de um objeto só pode ser modificado de maneiras permitidas e controladas;

**Benefícios do uso de métodos get e set:**

* **Encapsulamento:** Um dos princípios fundamentais da programação orientada a objetos é o encapsulamento, que trata de esconder detalhes de implementação e expor apenas operações seguras e controladas. Métodos get e set ajudam a atingir esse objetivo.
* **Manutenção de Código:** Eles fornecem uma interface estável para acessar e modificar dados. Se a implementação interna muda, o código que usa essa classe não precisa mudar, desde que a interface dos métodos get e set permaneça a mesma.
* **Controle:** Os métodos get e set fornecem controle total sobre como os atributos são lidos e modificados. Isso pode ser usado para implementar lógica de negócios, validações, registros e outras funções sempre que um atributo é acessado ou modificado.

**ENCAPSULAMENTO:**

* Não é viável ter acesso a modificar o atributo diretamente;
* Esconder os detalhes de implementação de uma classe, e expor apenas uma interface pública para interagir com ela.
* Quando o modificador de acesso do atributo é mudado para private, pode-se alterar a variável somente em um local específico. Nesse caso deve-se criar um método para permitir que alguém de fora possa registrar a característica.

public class Pessoa{

// definindo os atributos (características)

private String nome;

int idade;

// set: cadastra a característica

public void setNome(String nome) {

this.nome = nome;

}

// get: mostra na tela a característica

// Usar this quando se referencia ao atributo da classe.

public String getNome(){

return this.nome;

}

}

// No PrincipalPessoa:

umaPessoa.setNome("João");

System.out.println(umaPessoa.getNome());

´´´

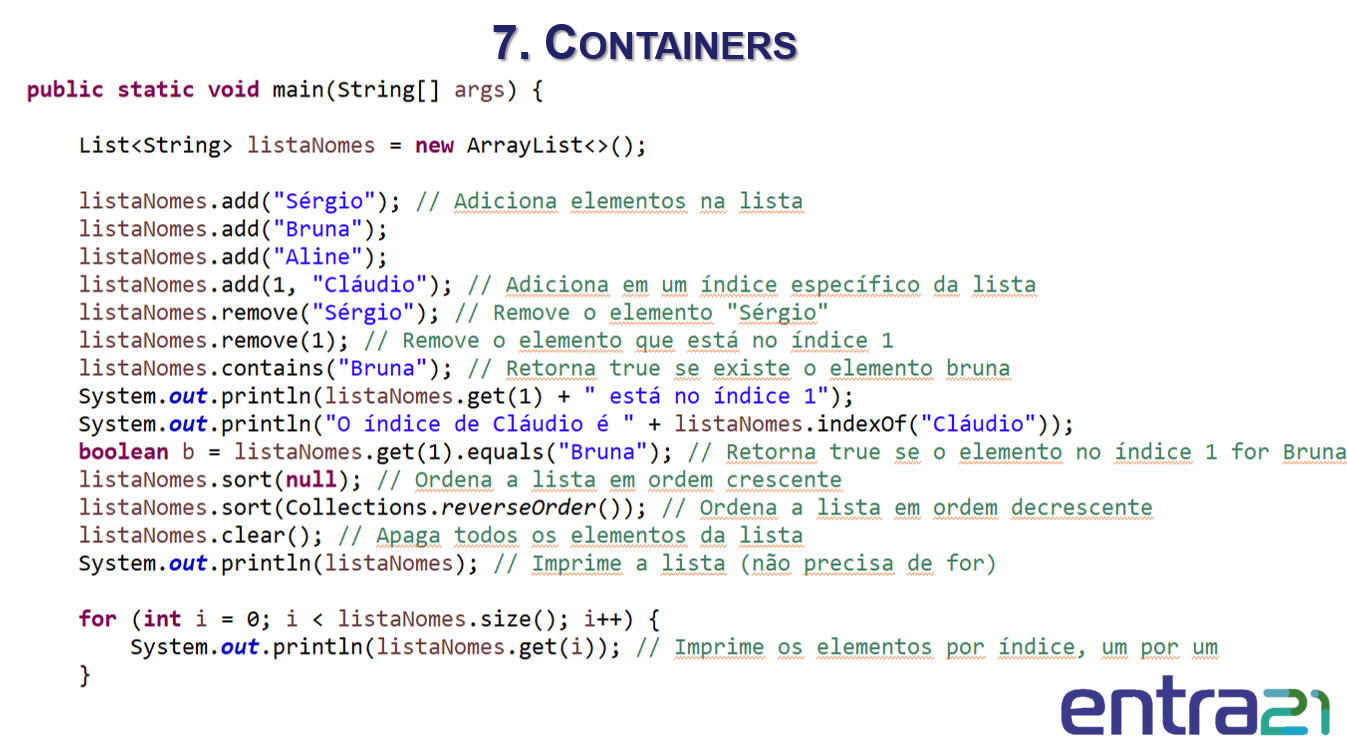
* Criar atributos: Botão direito -> Source action -> Generate getters e setters

**12/09:**

* Collection: há vários métodos que já foram implementados/prontos (são classes bases) e que outras classes utilizam (set, map, métodos)
* Classe abstrata não pode ser instanciada;
* List <- ArrayList (2 classes) (seta fechada = herança, ArrayList vem de List);

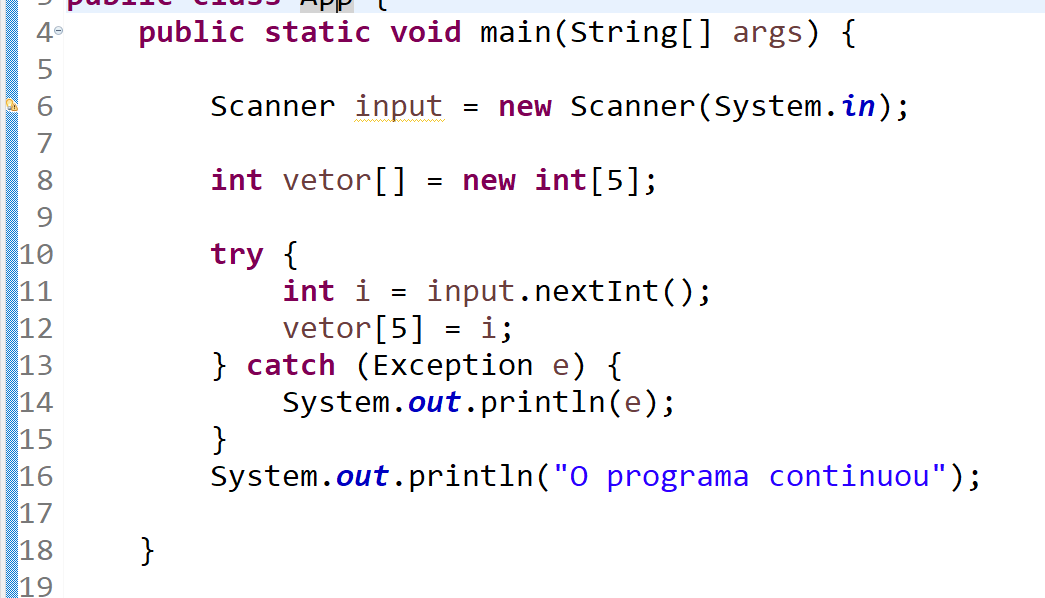
**ArrayList:**

* Normalmente inicia com 10 posições e ocupa mais espaço na memória;
* Quantidade armazenadas de elementos não é fixa, como nos Arrays (solução flexível para armazenar objetos);
* Objetos recuperados por sua posição;
* Armazena diferentes tipos de objetos;
* Não suporta primitivos (tem que ser empacotado – Integer, Double, Boolean, Float);
* Método add: adiciona um elemento na lista e retorna um booleano;
* ArrayList<**Float**> lista = new ArrayList; (< > operador diamante, mostra que o ArrayList só recebe números Float);
* Criamos uma List e instanciamos como tipo ArrayList;
* **depurar**: 2 cliques na linha + debug

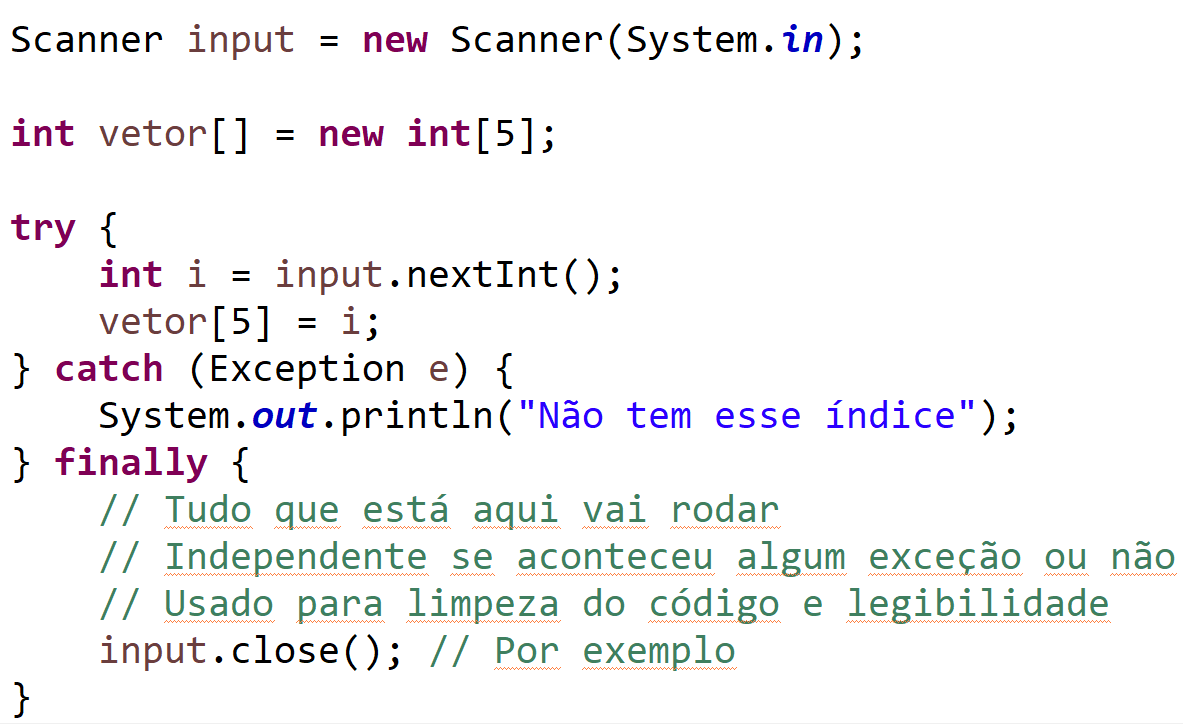


* Criar uma lista na classe do objeto;
* Static: atributo não é instanciado quando a classe for instanciada;
* public static List<Veiculo> listaVeiculo = new ArrayList<>();
* listaVeiculo não é parâmetro do construtor e não recebe getters e setters;
* for (Veiculo v : Veiculo.listaVeiculo):

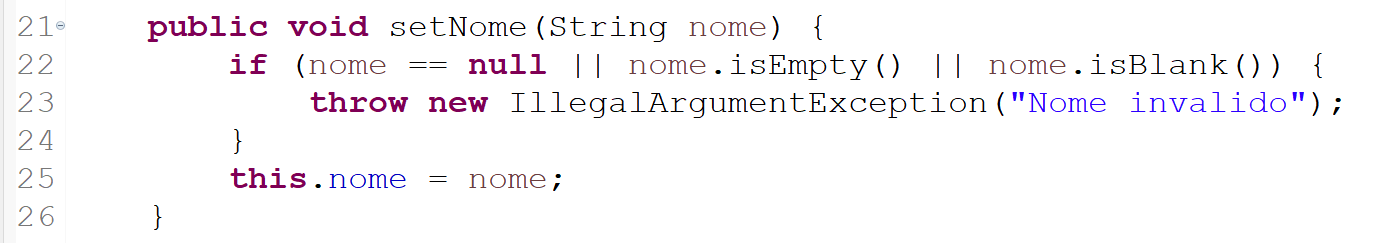
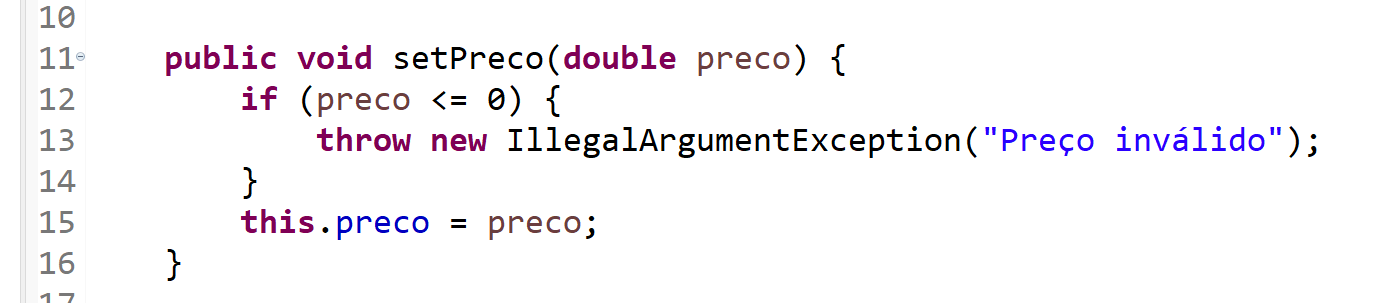
* **Exceção:**
* Exceção != erro (não conseguimos lidar com erros na base do código);
* Trechos de código escritos errados são exceções;
* Podemos fazer um catch nela para que o programa não pare:
* **try-catch** (para situações específicas): tenta executar o código, caso haja alguma exceção, ele utiliza o catch para lançar uma exceção mostrando o tipo do erro e após isso continua o código;



* **try-catch-finally:** o finally sempre roda, independentemente de haver erro ou não;



* **throw:** throw new IllegalArgumentException(“Nome inválido”); - programa para quando o valor inserido estiver errado (sempre verifica-se o null primeiro);



* **IndexOfAboutException:** para o programa quando o valor estiver errado
* **isEmpty():** verifica se o lenght é 0;
* **isEmpty().trim() ou isBlank():** retira os espaços vazios e verifica se é vazio;

**19/09:**

**HashMap:**

* **HashMap -> colletion: Map <Key,Value> (chave, valor associado);**
* A chave é o identificador único (nome, cidade);
* O valor é um list (idade, estado);
* CRUD(L) (Create, Read (1), Update, Delete, Lista);
* Liga uma chave a um valor (o valor é definido para a chave e para o valor);
* Mapas são um tipo de coleção que armazenam pares de objetos;
* Utiliza a chave (objeto) para pesquisar o valor (objeto) desejado;
* Objetos não tem uma ordem;
* Chaves são únicas (elemento da coleção set) e valores podem ser duplicados, assim, cada chave aponta pra um valor (duas chaves podem apontar para um mesmo valor, mas dois valores não podem ser apontados pela mesma chave);
* Permite que se faça os mesmos comandos que o ArrayList, mas sem o uso de vários ifs;
* Processamento do ArrayList é mais rápido que o HashMap;
* ArrayList garante a ordem dos elementos excluídos, o HashMap não garante;
* map.KeySet().size = retorna a quantidade de chaves
* map.Values().size = retorna a quantidade de valores
* map.size() = retorna o número de chaves pares-valor;
* Armazena itens em pares "chave / valor" que podem ser acessados por um índice de outro tipo (String).
* Um objeto é usado como chave (índice) para outro objeto (valor). Pode armazenar diferentes tipos: String chaves e Integer valores, ou do mesmo tipo, como: String chaves e String valores:
* Adicionando elementos (put):

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

System.out.println("Initial HashMap: " + numbers);

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("HashMap after put(): " + numbers);

}

SAÍDA:

HashMap inicial: {}

HashMap após put(): {Um=1, Dois=2, Três=3}

* ACESSANDO ELEMENTOS (get):

languages.put(1, "Java");

languages.put(2, "Python");

languages.put(3, "JavaScript");

System.out.println("HashMap: " + languages);

// return set view of keys

// using keySet()

System.out.println("Keys: " + languages.keySet());

// return set view of values

// using values()

System.out.println("Values: " + languages.values());

// return set view of key/value pairs

// using entrySet()

System.out.println("Key/Value mappings: " + languages.entrySet());

}

}

SAÍDA:

HashMap: {1=Java, 2=Python, 3=JavaScript}

Chaves: [1, 2, 3]

Valores: [Java, Python, JavaScript]

Mapeamentos de chave/valor: [1=Java, 2=Python, 3=JavaScript]

* ALTERANDO ELEMENTOS (raplace):

public static void main(String[] args) {

HashMap<Integer, String> languages = new HashMap<>();

languages.put(1, "Java");

languages.put(2, "Python");

languages.put(3, "JavaScript");

System.out.println("Original HashMap: " + languages);

// change element with key 2

languages.replace(2, "C++");

System.out.println("HashMap using replace(): " + languages);

}

SAÍDA:

HashMap original: {1=Java, 2=Python, 3=JavaScript}

HashMap usando replace(): {1=Java, 2=C++, 3=JavaScript}

* REMOVENDO ELEMENTOS (remove):

public static void main(String[] args) {

HashMap<Integer, String> languages = new HashMap<>();

languages.put(1, "Java");

languages.put(2, "Python");

languages.put(3, "JavaScript");

System.out.println("HashMap: " + languages);

// remove element associated with key 2

String value = languages.remove(2);

System.out.println("Removed value: " + value);

System.out.println("Updated HashMap: " + languages);

}

SAÍDA:

HashMap: {1=Java, 2=Python, 3=JavaScript}

Valor removido: Python

HashMap atualizado: {1=Java, 3=JavaScript}

* ITERANDO:

public static void main(String[] args) {

HashMap<Integer, String> languages = new HashMap<>();

languages.put(1, "Java");

languages.put(2, "Python");

languages.put(3, "JavaScript");

System.out.println("HashMap: " + languages);

System.out.print("Keys: ");

for (Integer key : languages.keySet()) {

System.out.print(key);

System.out.print(", ");

}

System.out.print("\nValues: ");

for (String value : languages.values()) {

System.out.print(value);

System.out.print(", ");

}

System.out.print("\nEntries: ");

for (Entry<Integer, String> entry : languages.entrySet()) {

System.out.print(entry);

System.out.print(", ");

}

}

SAÍDA:

HashMap: {1=Java, 2=Python, 3=JavaScript}

Chaves: 1, 2, 3,

Valores: Java, Python, JavaScript,

Entradas: 1=Java, 2=Python, 3=JavaScript,

* CRIANDO UM HASHMAP A PARTIR DE OUTRO:

public static void main(String[] args) {

// create a treemap

TreeMap<String, Integer> evenNumbers = new TreeMap<>();

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

System.out.println("TreeMap: " + evenNumbers);

// create hashmap from the treemap

HashMap<String, Integer> numbers = new HashMap<>(evenNumbers);

numbers.put("Three", 3);

System.out.println("HashMap: " + numbers);

}

SAÍDA:

TreeMap: {Quatro = 4, Dois = 2}

HashMap: {Dois=2, Três=3, Quatro=4}

**26/09:**

**RELACIONAMENTO ENTRE OBJETOS:**

* Atributo não é calculável;
* **Relacionamento temporário (dependência):**
* Um objeto necessita de outro objeto apenas durante a execução de suas responsabilidades;
* Não cria vínculos;
* Ocorre entre dois objetos **quando temporariamente um necessita/depende do outro**;
* Representação: **linha pontilhada (lado ao qual pertence) com seta aberta (lado dependente)**;
* EX: pegar um mouse **emprestado e devolver** no final da aula;
* Dependência por **declaração local:**

public void limpar() {

Apagador a = new Apagador();

a.esfregar();

}

* Dependência por **parâmetro:**

public void limpar(Apagador a) {

a.esfregar();

}

* **Relacionamento duradouro:**
* Durante a vida de um objeto, ele está ligado a outro para cumprir suas responsabilidades;
* **Associação:**
* Características:

1. Ter um **atributo do tipo classe** (Pessoa umaPessoa);
2. Ter um método que vincula o objeto ao atributo (EX: settar uma informação dentro do atributo);

* Objetos podem usar outros objetos;
* Deve-se nomear o relacionamento;
* Um atributo que remete outra classe e complementa o contexto da classe;
* Representação: **linha contínua (lado ao que pertence) com seta aberta (lado dependente)**;
* EX: **comprar um mouse, criando uma vinculação** com ele;
* EX: Pessoa usa anel, Trem usa ferrovia;
* EX: chamar a classe Endereco na classe Pessoa:

Private Endereco endereço;

* **Agregação:**
* Um objeto pode conter outro objeto;
* O todo existe sem as partes;
* Os objetos servem para complementar o todo;
* **Objeto existe sem as partes e as partes existem sem o todo**, são independentes;
* Representação: **Losango (lado ao qual pertence) com linha (lado dependente)**;
* EX: Carro possui motor e pneu, Sala possui mesa e cadeira;
* EX: um computador não deixa de ser computador se não possuir placa de vídeo ou HD;
* **Composição:**
* Um objeto pode ser **formado por outros objetos**;
* **O todo precisa das partes e as partes precisam do todo** (alta dependência entre parte e todo);
* Representação: **Losango preenchido (lado ao qual pertence) com linha (lado dependente)**;
* EX: Casa, livro, jardim, quarto;
* EX: A sala só existe porque existe um prédio e um andar;

**10/10:**

* **Sobrecarga do método:**
* Mesmo método, mas com número diferente de parâmetros (mesmo nome, lista de parâmetros diferentes);
* Assinatura <modificador> <retorno> <nomeMetodo> (<lista parâmetro>)
* EX 1:

public double somar(int num1, int num2) {

}

* EX 2:

public double somar(int num1, int num2, int num3) {

}

* **Sobrescrita:**
* Sobrescrever um método (igual o anterior – mesmos parâmetros);

**17/10:**

**HERANÇA**

* É a capacidade de especializar tipos de objetos (classes), de forma que os tipos especializados contenham, além de características estruturais e comportamentais já definidas por seus ancestrais, outras definidas por eles próprios;
* Permite a uma classe **herdar todo comportamento e atributos de outra classe**;
* **Utilizada** quando a **classe filha é do mesmo tipo que a classe pai**;
* **Vantagem:** não é necessário repetir as mesmas informações;
* Pode ser feita a partir de classes já construídas pelo programador ou por classes de terceiros ou classes
* **Forma:**
* tipo especializado ----(é um tipo de)---> tipo ancestral;
* Sintaxe:
* [subclasse a ser criada] extends [superclasse existente]
* Mostra que **todas as características da superclasse serão herdadas** pela subclasse;
* EX:

public class Veiculo() {

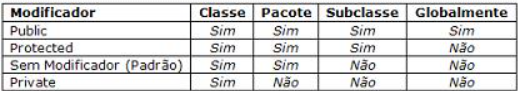
private double valor;

}

public VeiculoCarga extens Veiculo() {

}

* **Classe pai:** superclasse, classe base;
* **Classe filha:** subclasse, classe derivada;
* Classe filha (mais específica) herda atributos e métodos da classe pai (mais geral), possui atributos e métodos próprios;
* Representação: **linha (lado pertencedor) com seta FECHADA (lado ao que pertence)**;
* Modificadores de acesso:
* Na subclasse, deve-se primeiro **garantir a** **existencia/declaração das superclasses**. Para isso, usa-se o super(atributos) dentro do construtor



**ABSTRAÇÃO:**

* Não pode-se instanciar uma classe abstrata;
* Classe pai, pois não pode-se instanciá-la;
* Itálico no nome da classe abstrata e nos métodos abstratos;
* Métodos abstratos só são utilizados em classes abstratas;
* O método abstrato não tem corpo, e todas as classes filhas irão recebe-lo;
* Os métodos abstratos possuem implementação diferente para cada classe filha

**Lista estática**

* são acessadas apenas pela classe;
* feitas na classe pessoa;
* facilita a comparação na classe principal;
* EX: existem vários alunos, portanto, cria-se uma lista de alunos na classe aluno;

**Lista como atributo**

* são acessadas a partir de um objeto;
* raramente estão na mesma classe que o tipo delas;
* comparação pode ser feita na classe em que elas são atributos;
* Não vira parâmetro no construtor, utiliza-se um método próprio para adicionar objetos na lista;
* Possuem get e set;
* EX: um aluno possui várias notas, portanto, cria-se uma lista de notas na classe aluno;

**23/10:**

**POLIMORFISMO**

* **Um mesmo tipo de dado pode ser duas coisas**, apenas saberemos qual o tipo de dado após a instancia;
* uma mesma operação pode ser realizada por diferentes objetos de classes diferentes.
* Os construtores podem ter instancias diferentes;
* É **sobrescrita de métodos** (a assinatura do método necessariamente tem que ser igual: nome, parâmetros e tipo/retorno), quando o código descobre qual tipo de objeto ele é, e acaba tendo comportamento de acordo com o seu tipo de objeto;
* EX: Você se veste conforme o ambiente em que você está (igreja, praia, escola...);
* **Sobrecarga** é quando a assinatura do método é DIFERENTE, o que não acontece no polimorfismo;
* A classe pai possui um método que será sobrescrito pelos mesmos métodos existentes nas classes filhas (utiliza-se abstração na superclasse para não precisar escrever o mesmo método em todas as subclasses);
* **Polimorfismo é quando um método abstrato é passado da classe pai para as classes filhas e em cada uma delas, assumindo diferentes formas;**
* Dois objetos são distintos mesmo possuindo as mesmas características, pois cada um possui a sua própria identidade (gêmeos);
* Situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem, dependendo do seu tipo de criação;
* Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos.  Em Java, o polimorfismo semanifesta apenas em chamadas de métodos.
* caracterizado pela sobrescrita de um método da superclasse a partir da subclasse

public class Teste {

public void fazAlgo() {

System.out.println("Este é o método da super classe");

}

}

public class NovoTeste extends Teste {

@Override

public void fazAlgo() {

System.out.println("Este é o método foi sobrescrito");

}

}

public static void main(String args[]){

Teste T = new Teste();

T.fazAlgo(); // Retorna "Este é o método da super classe"

TesteNovo = TN = new TesteNovo();

TN.fazAlgo(); // Retorna "Este método foi sobrescrito"

}

* É desta forma que ocorre o polimorfismo de sobrescrita. A superclasse é sobrescrita e quando a classe for instanciada e o método fazAlgo() for invocado, o compilador executará o as linhas de código presentes no método da classe filhas.
* Quando definimos uma herança, todos os métodos e atributos da superclasse são passados para a classe filha. Entretanto, se um método for sobrescrito, o método definido na classe pai não será executado mais. Caso um método não seja sobrescrito, a regra diz que será executado o método da classe pai.
* Outro detalhe final é que o Java não permite herança múltipla (uma classe herdando de várias classes). Mas ele permite que uma classe seja estendida quantas vezes o programador desejar.
* EX: Som dos animais de uma fazenda:
* Embora objetos sejam da mesma superclasse, vão agir de maneira diferente em algum aspecto. Terão várias(poli) formas diferentes (morfismo).  
   A saída é criar um método chamado 'som()' na superclasse (só o cabeçalho, como veremos nos próximos tutoriais) e em cada subclasse criar um método diferente, que caracterize cada bicho.  
   Se não fizéssemos isso e invocássemos os métodos: vaca.som(), cachorro.som(), gato.som(), todos iriam fazer o mesmo barulho.  
  Com o polimorfismo: vaca.som() faria a vaca mugir, cachorro.som() faria o cachorro latir e gato.som() faria o gato miar. Todos continuam sendo objetos da classe "Animal".  
   E quando chegar mais animais na fazenda, adicione o método som() (abstrato na classe pai) no animal, de modo que ele poderá agir conforme suas características.

**07/11:**

**TRATAMENTO DE EXCEÇÃO:**

* Erros lógicos: tem a ver com a forma com que o programador fez o código (EX: loop infinito, contas erras);
* Erros de execução: programa compila, mas os testes dão errados (EX: estourar um array, null point exception, por isso deve-se usar tratamento) – ocorre por entrada inadequada de dados, manipulação indevida de arquivos, operação aritmética ilegal, falta de memória, comando não atendido pelo periférico;
* Os erros são detectados pela JVM;
* TRY-CATCH: