VEM SER

Módulo 01 - Java + OO

Aula 05 - OO e *Collections*

Conteúdo da aula

- Sobrecarga de construtores;
- *super*;
- Modificador de acesso *protected*;
- *final;*
- *abstract*;
- Polimorfismo:
  - Sobrecarga;
  - Sobrescrita;
- *Collections:*
  - Listas;
  - Filas;
  - Pilhas;
  - Mapas.

Sobrecarga de construtores

Sobrecarga de construtores

Acontece quando temos várias situações possíveis ao instanciar um objeto, necessitando de um construtor específico para viabilizar cada necessidade.

Sobrecarga de construtores

```
public Aluno(String nome) {
  this.nome = nome;
}

public Aluno(String nome, int idade) {
  this(nome);
  this.idade = idade;
}

public Aluno(String nome, int idade, String cidade) {
  this(nome, idade);
  this.cidade = cidade;
}
```

super

super

A palavra-chave ***super*** no Java é usada para se referir à classe pai de uma classe derivada. Ela pode ser usada para acessar campos, métodos e construtores da classe pai.

Construtor

A classe Professor é filha da classe Funcionario. Na classe Professor, podemos fazer o seguinte:

```
public Professor(String nome, String id, double salarioBruto) {
  super(nome, id, salarioBruto);
}
```

Método

A classe mãe “Funcionario” possui o método calcularSalarioLiquido, estamos utilizando a keyword super para adicionar mais coisa ao método:

```
public void calcularSalarioLiquido() {
  super.calcularSalarioLiquido();
  System.out.println("teste");
}
```

Atributos

Também podemos acessar atributos da classe mãe:

```
public void retornarInformacoes() {
  System.out.println(super.nome);
}
```

protected

protected

O ***protected*** em Java é usado para tornar um membro acessível **a classes derivadas** e **à própria classe**.

Modificador *final*

Modificador *final*

- O modificador final no Java é usado para tornar um membro da classe (atributo, método ou classe) **imutável**;
- Isso significa que o valor do membro **não pode ser alterado depois de ter sido inicializado**.

Modificador *final*

- Em atributos (exemplo):

public final double TAXA = 1.20;

Modificador *final*

- Em classes, a classe que tiver o modificador *final* não poderá ser herdada (não pode ter filhos).
- Isso é feito para evitar que os dados e métodos da classe sejam modificados por classes derivadas.
- Exemplo:

```
public final class Banco {
  String nome;
  public static final double TAXA = 1.20;
}
```

Modificador *final*

- Em métodos, o modificador *final* faz com que o método não possa ser sobrescrito por classes derivadas.
- Exemplo:

```
public final void imprime() {
  System.out.println("Alguma coisa");
}
```

Classes abstratas

Classes abstratas

- Uma classe abstrata é uma classe que **não pode ser instanciada diretamente**;
- Ela é usada como um modelo para outras classes, que podem herdar dela e adicionar seus próprios métodos e atributos.

Classes abstratas

```
public abstract class Pagamento {
  String tipo;
  String data;
}
```

Polimorfismo

Polimorfismo

O polimorfismo é um conceito da programação orientada a objetos que permite que você chame **o mesmo método de diferentes maneiras**, dependendo do tipo do objeto que está sendo usado.

Existem dois tipos:

- Polimorfismo de sobrecarga (o mais comum);
- Polimorfismo de sobreposição / sobrescrita.

Sobrecarga

O tipo de polimorfismo de sobrecarga permite a existência de vários métodos de mesmo nome

Sobrecarga

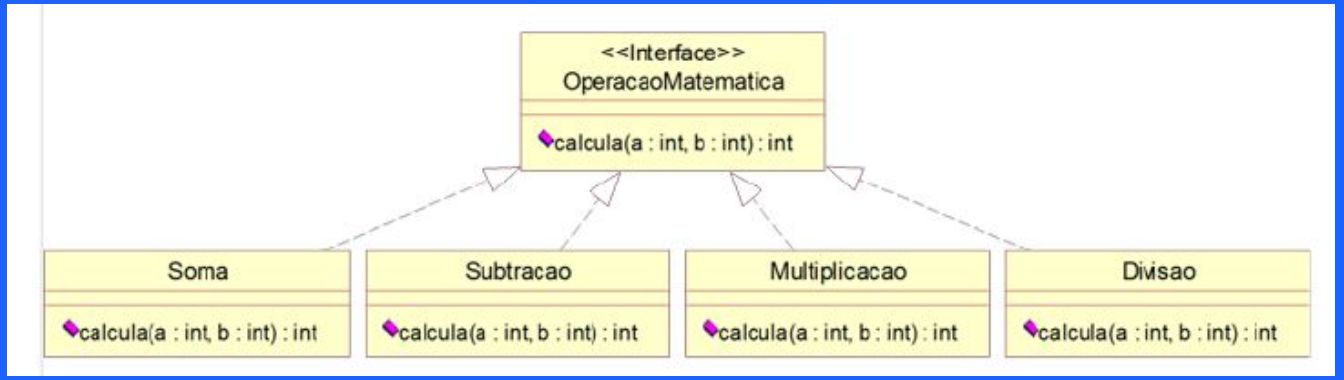
```
public void calcularSalarioLiquido() {
  System.out.println(this.salarioBruto - salarioBruto*0.1);
}

public void calcularSalarioLiquido(boolean ehMesDeFerias) {
  System.out.println(this.salarioBruto+((this.salarioBruto/3) - salarioBruto*0.1)*1.3);
}
```

Sobreposição

Acontece na herança, quando a subclasse (ou classe derivada) sobrepõe o método original.
A escolha de qual método será chamado depende do tipo do objeto que recebe a mensagem.

Vamos praticar!

Implemente o diagrama de classes abaixo com os seus respectivos métodos



- Após, sobrecarregue o método calcula em todas as classes com:
  - int calcula(a: double, b: double, c: double) e faça as operações de acordo com cada classe

Crie um método main para testar as operações.

DBC

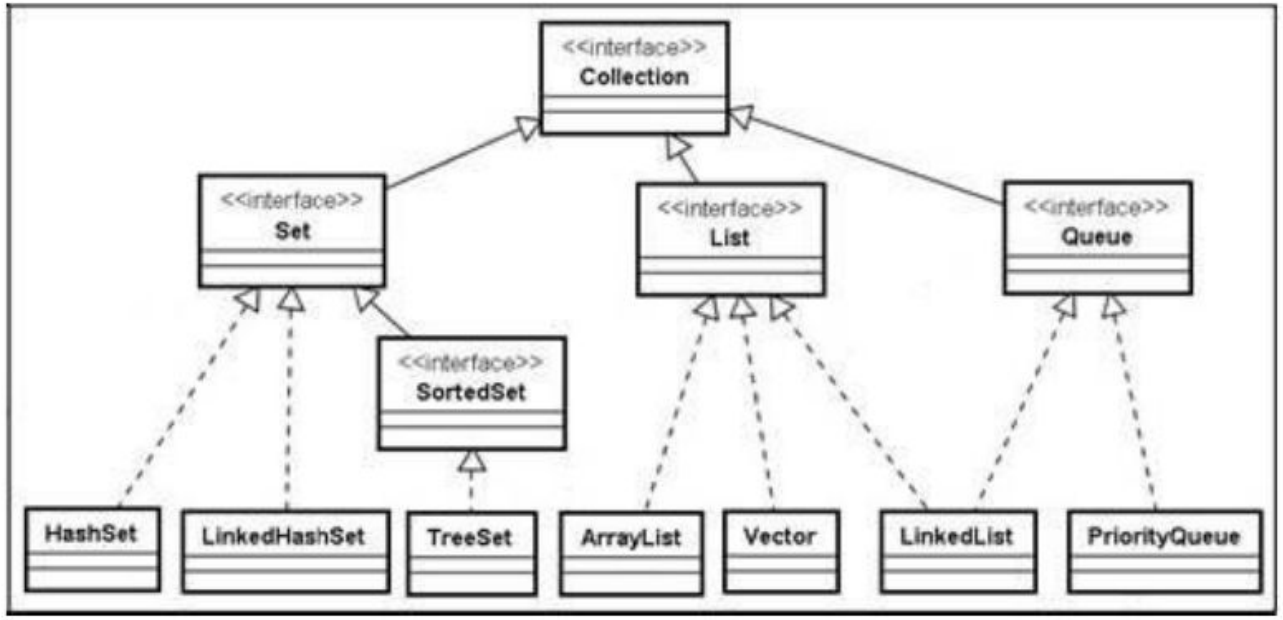
Collections

Collections

• Desde as primeiras versões, Java dispõe das estruturas de arrays e as classes Vector e Hashtable;
• A partir de Java 1.2, foi criado um conjunto de interfaces e classes denominado Collections Framework, que faz parte do pacote java.util;
• Collections Framework é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade.

Collections

- **Listas**: java.util.ArrayList
- **Filas**: java.util.LinkedList
- **Pilhas**: java.util.Stack
- **Mapas**: java.util.HashMap



Listas

FIFO (*first in, first out*): Primeiro a chegar, primeiro a sair.*

ArrayList (Listas)

```
ArrayList<String> lista = new ArrayList<>();
```

Métodos:

- add(elemento)
- remove(index)
- size()
- get(index)

Vamos praticar!

- Crie uma lista de nomes fixa;
- Imprima o penúltimo nome da lista;
- Imprima o primeiro nome da lista;
- Remova o último nome da lista;
- Ao final imprima todos os nomes e também a quantidade de nomes que
- sobraram na lista.

Filas

FIFO (*first in, first out*): Primeiro a chegar, primeiro a sair.

Queue (Filas)

```
Queue<String> fila = new LinkedList<>();
```

Métodos:

- add(elemento)
- poll()
- size()

Vamos praticar!

- Crie um sistema de senhas onde cada pessoa que chegar pegue uma senha numérica sequencial;
- Faça 5 pessoas entrarem na fila;
- Faça 2 pessoas serem atendidas;
- Faça 1 pessoa ser atendida;
- Faça mais 3 pessoas entrarem na fila;
- Faça 3 pessoas serem atendidas;
- Imprimir ao final a fila com todos os valores dela

Pilhas

LIFO (*last in, first out*): Último a entrar, primeiro a sair.

Stack (Pilhas)

```
Stack<String> pilha = new Stack<>();
```

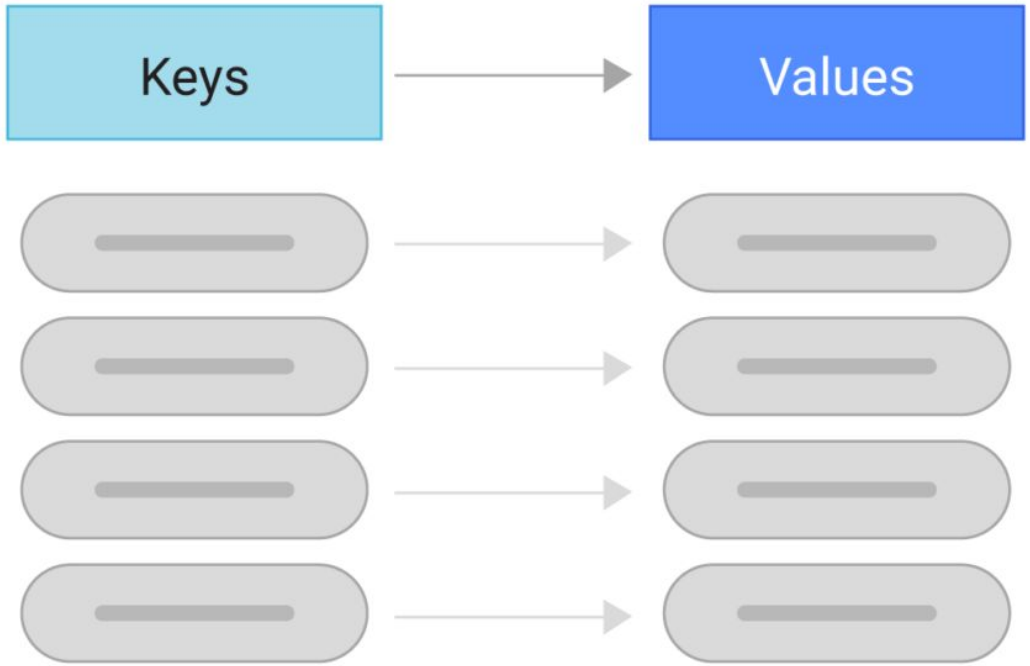
Métodos:

- push(elemento)
- peek()
- pop()
- size()

Vamos praticar!

- Crie um programa que leia 15 números e proceda, para cada um deles, como segue:
  - se o número for par, insira-o na pilha;
  - se o número lido for ímpar, retire um número da pilha;
  - Ao final, se ainda conter elementos, esvazie a pilha imprimindo os elementos.

Par chave e valor (*key-value pairs*)



Crédito da imagem: https://www.scylladb.com/glossary/key-value-database/

Map (Mapas)

```
Map<String, String> mapa = new HashMap<>();
```

Métodos:

- put(chave, valor)
- remove(chave)
- get(chave)

Vamos praticar!

- Crie um programa que leia o cpfs, o nomes de pessoas e adicione a um mapa.
- Em seguida, peça ao usuário para consultar um cpf e exiba o nome daquela pessoa com o cpf digitado e remove o cpf caso esteja cadastrado. Mostrar a mensagem que o cpf não existe caso não exista.
- Ao final imprima o conteúdo do Mapa.

Comparando listas

- A interface ***Comparator*** é usada para comparar objetos.
- Ela tem um único método chamado ***compare()***, que recebe dois objetos como parâmetros e retorna um inteiro.
- O valor retornado pelo método compare() indica qual objeto é maior, menor ou igual ao outro objeto.

Comparando listas

O método compare() deve retornar um dos seguintes valores:

- **Valores negativos**: O primeiro objeto é menor que o segundo objeto;
- **0**: Os dois objetos são iguais;
- **Valores positivos**: O primeiro objeto é maior que o segundo objeto.

Comparando listas

```
Comparator<Integer> comparator = new Comparator<Integer>() {
  @Override
  public int compare(Integer number1, Integer number2) {
    return number1 - number2;
  }
};
```

Comparando listas

```
Comparator<Pessoa> comparador = new Comparator<Pessoa>() {
  @Override
  public int compare(Pessoa pessoa1, Pessoa pessoa2) {
    return pessoa1.getName().compareTo(pessoa2.getName());
  }
};

Pessoa pessoa1 = new Pessoa("Maria");
Pessoa pessoa2 = new Pessoa("Pedro");

int resultado = comparador.compare(pessoa1, pessoa2);
```

Vamos praticar!

- Crie uma lista de nomes e idades (objeto Pessoa) (pode ser fixa) e ordene de forma crescente por nome e imprima os valores;
- Na sequência, ordene por idade (do mais velho para o mais novo) e imprima os valores;
- Por fim, ordene por idade e por nome respeitado o critério:
  - Nome crescente
  - Se nome for igual, ordena por idade

Task #1 individual

- Faça as correções referentes aos apontamentos das tasks passadas
- Copie a pasta "**conta-corrente2**" para "**conta-corrente3**" dentro da pasta **aula05** e siga as instruções abaixo:
- Transforme a classe Conta em *abstract*;
- Troque os atributos "arrays" da classe cliente para ArrayLists.
- Crie uma nova classe **ContaPagamento** que deverá estender de Conta (ser filha de Conta) e implementar a interface Impressao, porém com um atributo estático final TAXA_SAQUE com o valor de R$4,25;
- Sobrescreva o método sacar(valor) e faça descontar o valor da taxa do saldo da conta pagamento;
- Crie um método main, crie 2 clientes sendo:
  - 1 cliente com conta pagamento e conta corrente
  - 1 cliente com conta poupança
- Realize ao menos 3 movimentações entre as contas (saque, depósito e transferência)
- Ao final, imprima os dados da conta

Task #2 em grupo

- Elabore ao menos 2 ***Collections*** para o seu projeto (podem ser do mesmo tipo)

Referências

https://alissonraphaeloliveira.medium.com/algoritmos-e-estrutura-de-dados-introdu%C3%A7%C3%A3o-6770bd3e7586

https://dev.to/rodolfobueno/fixando-os-conceitos-de-fila-e-pilha-com-a-ajuda-do-bob-esponja-1d9g

https://www.youtube.com/playlist?list=PL62G310vn6nFIsOCC0H-C2infYgwm8SWW

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html


Let's Tech Up Together