Java Developer

Java Core

Sumário

Objetivos deste Módulo

Ao final deste módulo, objetiva-se que o aluno adquira:

Conhecimentos:

- Orientação a Objetos: overload
- Coleções:
  - List: ArrayList, LinkedList
  - Set: TreeSet, HashSet, LinkedHashSet
  - Map: TreeMap, HashMap, LinkedHashMap
  - Queue: LinkedList

1. Overload

A **sobrecarga** de um método (method **overload**) em Java é uma técnica que permite que uma classe tenha **vários métodos com o mesmo nome**, mas com **parâmetros diferentes**. Isso significa que você pode ter vários métodos em uma classe com o mesmo nome, mas que aceitam diferentes tipos ou números de argumentos.

Aqui estão os principais pontos sobre a sobrecarga de método em Java:

Nome de método igual: Os métodos sobrecarregados têm o mesmo nome, mas diferem em termos de assinatura, que inclui o nome do método, o número de parâmetros ou o tipo dos parâmetros.

Assinatura diferente: A assinatura de um método inclui o nome do método e a lista de tipos e ordem dos parâmetros. Dois métodos têm uma assinatura diferente se qualquer um dos seguintes critérios for atendido:

- O número de parâmetros é diferente.
- Os tipos de parâmetros são diferentes.
- A ordem dos tipos de parâmetros é diferente.
- Tipo de Retorno: A sobrecarga de método não é determinada apenas pelo tipo de retorno. Você não pode criar métodos sobrecarregados com a mesma assinatura, mas com tipos de retorno diferentes.
- Usos Comuns: A sobrecarga de método é comumente usada para criar métodos que executam operações similares, mas com diferentes tipos de entrada. Por exemplo, você pode ter um método soma que aceita dois inteiros e outro método soma que aceita dois números de ponto flutuante.

Exemplo:

```
public class Calculadora {

    // Alterar o retorno não garante a sobrecarga
    // se houver outro método com mesmo nome
    // e parâmetros similares
    /*public Integer soma(int a, int b) {
        return a + b;
    }*/

    public int soma(int a, int b) {
        return a + b;
    }

    public double soma(double a, double b) {
        return a + b;
    }

    public int soma(int a, int b, int c) {
        return a + b + c;
    }

    public String concatena(String str1, String str2) {
        return str1 + str2;
    }
}
```

```
public class ExemploSobrecarga {

    public static void main(String[] args) {

        Calculadora calculadora = new Calculadora();

        int resultado1 = calculadora.soma(5, 7);
        double resultado2 = calculadora.soma(3.5, 2.5);
        int resultado3 = calculadora.soma(1, 2, 3);
        String resultado4 = calculadora.concatena("Olá, ", "mundo!");

        System.out.println("Resultado 1: " + resultado1); // 12
        System.out.println("Resultado 2: " + resultado2); // 6.0
        System.out.println("Resultado 3: " + resultado3); // 6
        System.out.println("Resultado 4: " + resultado4); // Olá, mundo!
    }
}
```

Neste exemplo, a classe **Calculadora** possui vários métodos com o mesmo nome, mas com parâmetros diferentes. Isso é uma sobrecarga de método. Você pode chamar esses métodos com base nos argumentos que deseja passar.

Quando você chama um método sobrecarregado, o Java determina qual versão do método deve ser chamada com base nos argumentos passados. O Java realiza a resolução de sobrecarga em tempo de compilação, escolhendo o método apropriado com base na assinatura dos argumentos fornecidos.

A sobrecarga de método é uma maneira eficaz de tornar seu código mais flexível e conveniente, permitindo que você use o mesmo nome de método para realizar tarefas semelhantes em diferentes situações. Isso ajuda a melhorar a legibilidade e a manutenção do código, simplificando a nomenclatura dos métodos quando eles têm funcionalidades relacionadas.

IMPORTANTE!!!!

Override e overload são duas técnicas de programação em Java que envolvem a definição de métodos em classes, mas têm finalidades diferentes e são aplicadas em contextos distintos:

Override (Sobrescrita): ocorre quando uma classe filha (subclasse) ou implementação da interface fornece uma implementação específica para um método que já foi declarado. O método na classe filha ou implementação de interface deve ter o mesmo nome, tipo de retorno e lista de parâmetros que o método na classe pai ou interface.

Exemplo: Em uma hierarquia de classes de animais, você pode ter um método emitirSom() na classe pai Animal e, em seguida, cada classe de animal específica (como Cachorro e Gato) pode sobrescrever esse método para emitir sons específicos.

Overload (Sobrecarga): ocorre quando uma classe tem vários métodos com o mesmo nome, mas com diferentes assinaturas. A assinatura de um método inclui o nome do método, o número e/ou tipos de seus parâmetros.

Exemplo: Uma classe Calculadora pode ter vários métodos soma, como soma(int a, int b) para números inteiros e soma(double a, double b) para números de ponto flutuante. Ambos os métodos têm o mesmo nome, mas a assinatura é diferente.

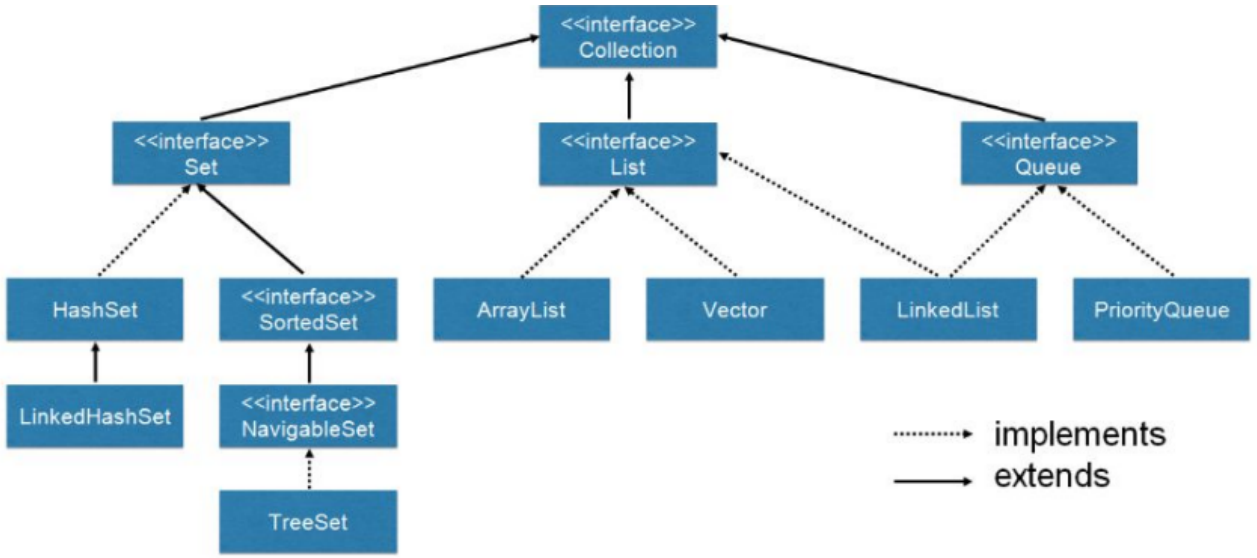
Em resumo, a principal diferença entre **override (sobrescrita)** e **overload (sobrecarga)** está na finalidade e no contexto:

A **sobrescrita** é usada para fornecer uma implementação específica de um método em uma classe filha que já foi definido na classe pai. Ou no caso de interface, a fornecer uma implementação específica da assinatura do método definido na interface que o mesmo implementa.

A **sobrecarga** é usada para ter vários métodos com o mesmo nome em uma classe, mas com assinaturas diferentes, para realizar tarefas semelhantes com tipos de entrada diferentes.

2. Coleções

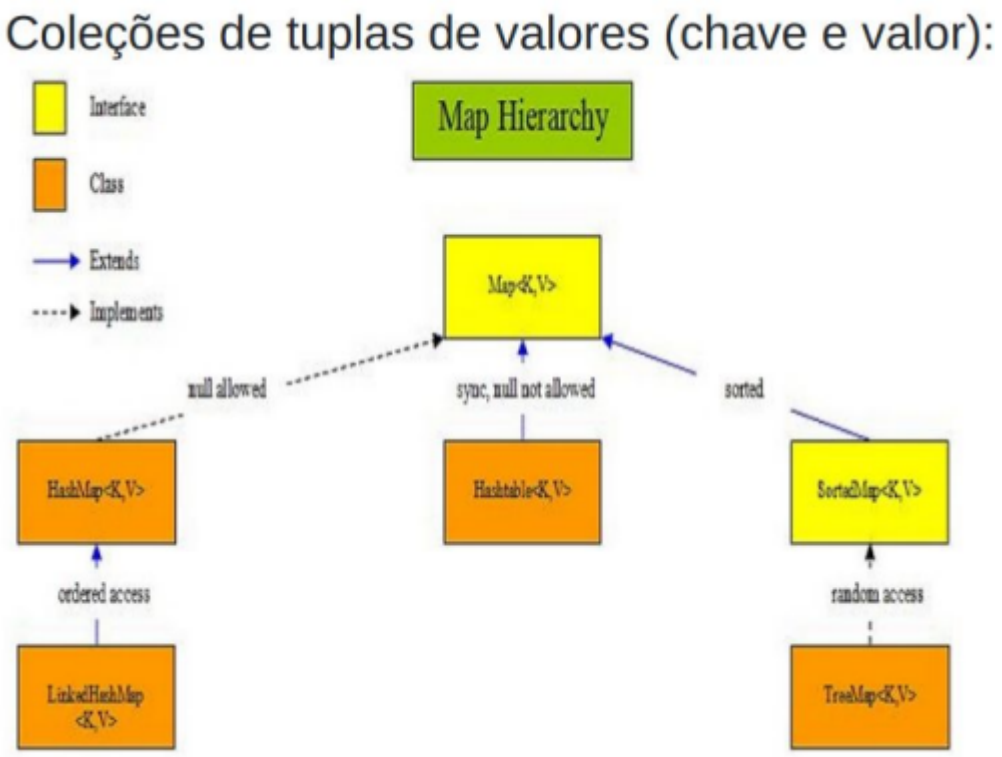
Interface Collection



Documentação oficial: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html

Não confundir com a classe Collections

Interface Map



Documentação oficial: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Map.html

Resumindo:

List: Aceita elementos duplicados
Set: Não aceita elementos duplicados
Map: Guarda coleções de tuplas de valores (chave e valor)
Queue: Trabalha com fila e pilhas

3. List

Interface List

Permite duplicações.

Principais implementações:

- ArrayList: Utiliza array internamente, mais rápido para acesso de leitura.
- LinkedList: Utiliza lista encadeada, mais rápida para inserção e deleção.

Exemplo:

```
import java.util.*;

public class ExemploList {

    public static void main(String[] args) {

        // List: Aceita elementos duplicados: TEM Java duas vezes na lista

        // Criando um LinkedList e populando
        List<String> linguagens = new LinkedList<>();

        linguagens.add("Java");
        linguagens.add("PHP");
        linguagens.add("PHP");
        linguagens.add("C#");
        linguagens.add("JS");
        linguagens.add("Scala");
        linguagens.add("Java");

        System.out.println(linguagens);

        /*System.out.println("Para LinkedList não é uma boa");
        for (int i = 0; i < linguagens.size(); i++) {
            System.out.println(linguagens.get(i));
        }*/

        //imprimirListaUsandoForeach(linguagens);

        //imprimirListaUsandoWhile(linguagens);

        //System.out.println("***********");
       // System.out.println("Métodos");
        //testarMetodosLista(linguagens);

        //Converter em outro tipo de lista
       /* Set<String> lings = new TreeSet<>(linguagens);
        System.out.println(lings);*/
    }
```

```
    private static void testarMetodosLista(List<String> linguagens) {

        // Na lista tem uma linguagem PHP? Usar método contains
        System.out.println("Contains: " + linguagens.contains("PHP"));

        // Qual posição da lista tem Java? -> indexOf (primeira ocorrência)
        System.out.println("IndexOf :" + linguagens.indexOf("Java"));

        // Qual posição da lista tem Java? -> lastIndexOf (última ocorrência)
        System.out.println("LastIndexOf: " + linguagens.lastIndexOf("Java"));

        // Como remover Java da lista? -> remove
        boolean removeuJava = linguagens.remove("Java");
        System.out.println("Remove Java " + removeuJava);

        boolean removeuJava2 = linguagens.remove("Java2");
        System.out.println("Remove Java2 " + removeuJava2);

        // Inserir na posição 0 a linguagem Kotlin
        linguagens.add(0, "Cobol");
        linguagens.set(0, "Kotlin");

        System.out.println("********************");
        imprimirListaUsandoForeach(linguagens);

        // Limpa a lista -> remove todos os elementos
        linguagens.clear();
        System.out.println("++++++++++++++++++");
        imprimirListaUsandoForeach(linguagens);

    }

    private static void imprimirListaUsandoWhile(List<String> linguagens) {
        System.out.println("=========");
        Iterator<String> iterator = linguagens.iterator();
        while (iterator.hasNext()) {
            //i++
            //numeros[i]
            String ling = iterator.next();
            System.out.println(ling);
        }
/*
        Iterator -> Precisamos de um objeto que permita iterar na lista
        iterator.hasNext -> verifica se tem um próximo elemento
        next -> acessa o elemento, similar a numeros[i]
*/
    }

    private static void imprimirListaUsandoForeach(List<String> linguagens) {
        //System.out.println("########");
        for (String ling: linguagens) {
            System.out.println(ling);
        }
    }
}
```

4. Set

Interface Set

Não permite duplicações.

Principais implementações:

- TreeSet: Ordena automaticamente os elementos de acordo com seu valor.
- HashSet: Não garante ordenação.
- LinkedHashSet: Mantém ordem de inserção.

Exemplo:

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class ExemploSet {

    public static void main(String[] args) {

        // Set: Não aceita elementos duplicados

        //TreeSet: Ordena automaticamente os elementos de acordo com seu valor
        //listarUsandoTreeSet();

        // HashSet: Não garante ordenação
        //listarUsandoHashSet();

        // LinkedHashSet: Mantém ordem de inserção
        listarUsandoLinkedHashSet();

    }

    private static void listarUsandoHashSet() {
        System.out.println("- FRUTAS - ");

        Set<String> frutas = new HashSet<>();

        frutas.add("banana");
        frutas.add("maçã");
        frutas.add("manga");
        frutas.add("limão");
        frutas.add("pera");
        frutas.add("morango");
        frutas.add("morango");
        frutas.add("melão");
```

```
        for (String fruta : frutas) {
            System.out.println(fruta);
        }
    }

    private static void listarUsandoLinkedHashSet() {

        System.out.println("- PESSOAS -");

        Set<String> pessoas = new LinkedHashSet<>();

        pessoas.add("Thiago");
        pessoas.add("Jhonny");
        pessoas.add("Maria");
        pessoas.add("Paulo");
        pessoas.add("Daniel");
        pessoas.add("Alice");
        pessoas.add("Thiago");

        for (String pessoa : pessoas) {
            System.out.println(pessoa);
        }
    }

    private static void listarUsandoTreeSet() {

        System.out.println("- FABRICANTES -");

        Set<String> fabricantes = new TreeSet<>();

        fabricantes.add("Dell");
        fabricantes.add("HP");
        fabricantes.add("Dell");
        fabricantes.add("Lenovo");
        fabricantes.add("Apple");

        for (String fabricante : fabricantes) {
            System.out.println(fabricante);
        }

        Set<Integer> numeros = new TreeSet<>();
        numeros.add(100);
        numeros.add(20);
        numeros.add(5);
        numeros.add(1000);

        for (Integer numero : numeros) {
            System.out.println(numero);
        }
    }
}
```

5. Map

Interface Map

Guarda coleções de tuplas de valores (chave/key e valor/value).

Principais implementações:

- TreeMap: Registros são ordenados ascendentes pela sua chave.
- HashMap: Não garante ordenação.
- LinkedHashMap: Mantém a ordem de inserção.

Exemplo:

```
import java.util.LinkedHashMap;
import java.util.Map;

public class ExemploMap {

    public static void main(String[] args) {

        // Map: Guarda coleções de tuplas de valores (chave e valor)
        Map<String, String> capitais = new LinkedHashMap<>();

        capitais.put("RS", "Porto Alegre");
        capitais.put("SC", "Floripa");
        capitais.put("SP", "São Paulo");
        capitais.put("MG", "Belo Horizonte");
        capitais.put("PR", "Curitiba");
        capitais.put("PR", "Curitibanos");
        capitais.put("AC", "Rio Branco");
        capitais.put(null, "abc");
        capitais.put(null, "def");

        //listarKeys(capitais);

        //listarValues(capitais);

        listarKeyValue(capitais);

        System.out.println("Capital de MG: " + capitais.get("MG"));

        String rj = capitais.get("RJ");
        System.out.println("RJ " + rj);

        // testando antes de inserir
        if(rj == null) {
            capitais.put("RJ", "Rio de Janeiro");
        }

        rj = capitais.get("RJ");
        System.out.println("RJ " + rj);
```

```
    }

    private static void listarKeyValue(Map<String, String> capitais) {

        System.out.println("Chave/valor (Entry):");

        for (Map.Entry<String, String> registro : capitais.entrySet()) {
            System.out.printf("Capitais: %s - %s \n", registro.getKey(), registro.getValue());
        }
    }

    private static void listarValues(Map<String, String> capitais) {
        System.out.println("Cidade:");
        for (String value : capitais.values()) {
            System.out.println(value);
        }
    }

    private static void listarKeys(Map<String, String> capitais) {
        System.out.println("Estado:");
        for (String key : capitais.keySet()) {
            System.out.println(key);
        }
    }
}
```

6. Queue

Interface Queue

Queue é uma estrutura de dados de lista em Java seguindo a ordem de inserção *First In First Out* (FIFO) ou seja, Primeiro que entra, primeiro que sai. Exatamente como uma fila de pessoas.

Principais implementações:

- LinkedList

Exemplo:

```
import java.util.LinkedList;
import java.util.Queue;

public class ExemploQueue {

    public static void main(String[] args) {

        Queue diasSemana   = new LinkedList();

        diasSemana.add("Sunday");
        diasSemana.add("Monday");
        diasSemana.add("Tuesday");
        diasSemana.add("Wednesday");
        diasSemana.add("Thursday");
```

```
        diasSemana.add("Friday");
        diasSemana.add("Saturday");

        System.out.println("Dias da semana: \t" + diasSemana + "\n");

        // q.poll() : remove e retorna o início da fila
        System.out.println("poll() retornou: " + diasSemana.poll());
        System.out.println("Dias da semana atualizado!\t" + diasSemana + "\n");

        System.out.println("poll() retornou: " + diasSemana.poll());
        System.out.println("Dias da semana atualizado!\t" + diasSemana + "\n");

        //q.peek() : não remove, mas apenas retorna o início da fila
        System.out.println("peek() retornou: " + diasSemana.peek());
        System.out.println("Dias da semana atualizado!\t" + diasSemana + "\n");

        // q.remove() : remove e retorna o início da fila
        System.out.println("remove() retornou: " + diasSemana.remove());
        System.out.println("Dias da semana atualizado!\t" + diasSemana + "\n");

        //remove() e poll() são exatamente iguais. Eles funcionam de forma idêntica em circunstâncias
normais.
        // Mas quando a fila está vazia, remove() lança NoSuchElementException , enquanto poll retorna
null

    }
}
```

7. Exercícios

1) Crie uma coleção de Strings vazia chamada "pessoas".

   Adicione os nomes: Bryana, Kiersten, Zaneta, Frank, Bryana, Tedi, Marigold, Devan, Jerrilyn, Isac, Kathrine.

   Imprima a lista completa.

   Crie uma nova coleção de Strings chamada "primeiros".

   Nesta nova lista, não pode haver repetição, deve estar ordenada e deve conter somente os primeiros 5 nomes.

   Imprima a lista de "primeiros".

2) A partir da seguinte lista de países campeões da copa:

   ```
   List<String> campeoes = new ArrayList<String>();

   campeoes.add("1930 - Uruguai");
   campeoes.add("1934 - Itália");
   campeoes.add("1938 - Itália");
   campeoes.add("1950 - Uruguai");
   campeoes.add("1954 - Alemanha");
   campeoes.add("1958 - Brasil");
   ```

```
campeoes.add("1962 - Brasil");
campeoes.add("1966 - Inglaterra");
campeoes.add("1970 - Brasil");
campeoes.add("1974 - Alemanha");
campeoes.add("1978 - Argentina");
campeoes.add("1982 - Itália");
campeoes.add("1986 - Argentina");
campeoes.add("1990 - Alemanha");
campeoes.add("1994 - Brasil");
campeoes.add("1998 - França");
campeoes.add("2002 - Brasil");
campeoes.add("2006 - Itália");
campeoes.add("2010 - Espanha");
campeoes.add("2014 - Alemanha");
campeoes.add("2018 - França");
```

Crie um map chamado “quantidadeTitulos” onde a key é o país e o valor é a quantidade de vezes que foi campeão.

Percorra a lista criada, adicionando ao Map conforme achar necessário.

Imprima o país e a quantidade de copas. O ano pode ser ignorado.

3) Crie um programa que receba uma lista de números inteiros e remova todos os elementos pares da lista.

4) Crie um programa que encontre o maior elemento em uma lista de números inteiros.

5) Crie um programa que receba duas listas de strings e as una (mescle) em uma única lista.