



Usando Collection - Parte 2

H

ashMap / Hashtable, LinkedHashMap e TreeMap

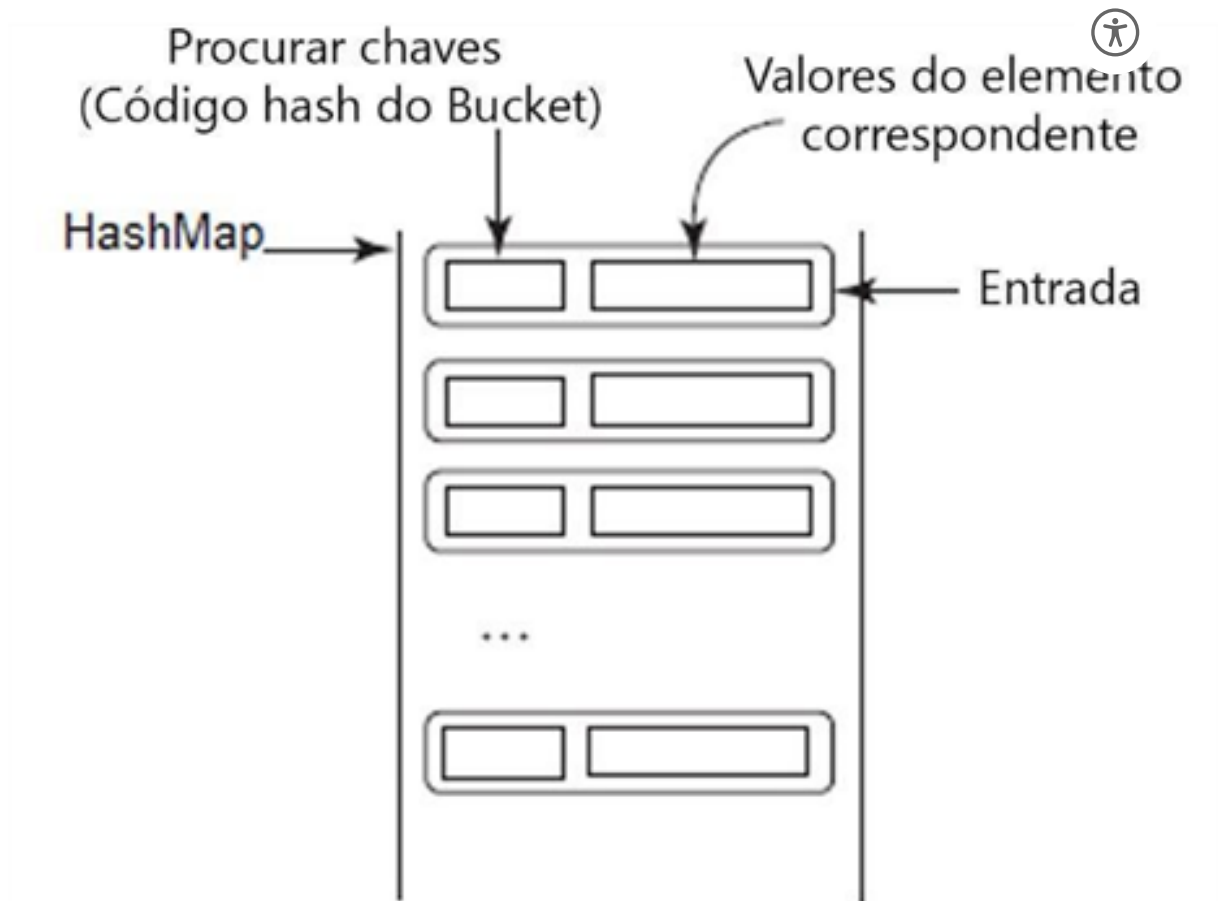
A ideia básica de um mapa é que ele mantenha associações de valores-chave (pares) para que você possa procurar um valor usando uma chave.

HashMap / Hashtable

O HashMap possui implementação baseada em uma tabela de hash. (Use essa classe em vez do Hashtable, que é uma classe herdada). O HashMap fornece um mapa não classificado e não ordenado. Quando você precisa de um mapa e não se importa com o pedido (quando itera), o HashMap é a escolha certa. As chaves do HashMap são como Set significa que nenhuma duplicação é permitida e desordenada, enquanto os valores podem ser qualquer objeto, mesmo nulo ou duplicado, também é permitido. O HashMap é muito semelhante ao Hashtable. A única diferença é que o Hashtable tem todo o método sincronizado para segurança do encadeamento, enquanto o HashMap possui métodos não sincronizados para obter melhor desempenho.

Podemos visualizar o HashMap como o diagrama abaixo, onde temos as chaves conforme o código hash e os valores correspondentes.





O HashMap fornece desempenho em tempo constante para inserir e localizar pares. O desempenho pode ser ajustado por meio de construtores que permitem definir a capacidade e o fator de carga da tabela de hash.

Construtores HashMap

HashMap ()

Construtor HashMap padrão (com capacidade padrão de 16 e fator de carga 0,75)

HashMap(Map<? extends KeyObject, ? extends ValueObject> m)

Isso é usado para criar o HashMap com base na implementação de mapa existente m.

HashMap(int capacidade)



Isso é usado para inicializar o HashMap com capacidade e fator de carga padrão.

HashMap(int capacidade, float carregaFator)

Isso é usado para inicializar o HashMap com capacidade e fator de carga personalizado.

As operações básicas do HashMap (put, get, containsKey, containsValue, size e is Empty) se comportam exatamente como suas contrapartes no Hashtable. O HashMap possui o **método toString ()** substituído para imprimir os pares de valores-chave facilmente. O programa a seguir ilustra o HashMap. Mapeia nomes para salário. Observe como uma visualização de conjunto é obtida e usada.

```
import java.util.*;
```

```
public class ArmazenaSalarioFuncionario {
```

```
    public static void main(String[] args) {
```

```
        //Abaixo da linha criará o HashMap com tamanho inicial 10 e  
        0,5 fator de carga
```

```
        Map<String, Integer>funcSal = new HashMap<String,  
        Integer>(10, 0.5f);
```

```
        //Adicionando nome do funcionário e salário ao map
```

```
        funcSal.put("Rita", 10000);
```



```
funcSal.put("Samuel", 20000);
```

```
funcSal.put("Maria", 30000);
```

```
funcSal.put("Nicole", 1000);
```

```
funcSal.put("Noemi", 15000);
```

```
funcSal.put("Raquel", 10000); // Valor duplicado também  
permitido, mas as chaves não devem ser duplicadas
```

```
funcSal.put("Nicolas", null); //O valor também pode ser nulo
```

```
System.out.println("Original Map: "+ funcSal); // Mostrando  
Map completo
```

```
//Adicionando novo funcionário ao Mapa para ver a ordem  
das alterações nos objetos
```

```
funcSal.put("Renato", 23000);
```

```
//Removing one key-value pair
```

```
funcSal.remove("Nicolas");
```

```
System.out.println("Updated Map: "+funcSal); // Mostrando  
Map completo
```

```
//Imprimir todas as chaves
```

```
System.out.println(funcSal.keySet());
```

```
//Imprimindo todos os valores
```

```
System.out.println(funcSal.values());
```

}



}

Resultado:

```
Map original: {Samuel=20000, Nicolas=null, Raquel=10000, Nicole=1000,
Noemi=15000, Rita=10000, Maria=30000}
Map atualizado: {Renato=23000, Samuel=20000, Raquel=10000, Nicole=1000,
Noemi=15000, Rita=10000, Maria=30000}
[Renato, Samuel, Raquel, Nicole, Noemi, Rita, Maria]
[23000, 20000, 10000, 1000, 15000, 10000, 30000]
```

Java LinkedHashMap

LinkedHashMap estende o HashMap. Ele mantém uma lista vinculada das entradas no mapa, na ordem em que foram inseridas. Isso permite a iteração da ordem de inserção no mapa. Ou seja, ao iterar por meio de uma exibição de coleção de um LinkedHashMap, os elementos serão retornados na ordem em que foram inseridos. Além disso, se alguém inserir a chave novamente no LinkedHashMap, os pedidos originais serão mantidos. Isso permite a iteração da ordem de inserção no mapa. Ou seja, ao iterar um LinkedHashMap, os elementos serão retornados na ordem em que foram inseridos. Você também pode criar um LinkedHashMap que retorne seus elementos na ordem em que foram acessados pela última vez.

Construtores

LinkedHashMap ()

Esse construtor constrói uma instância do LinkedHashMap ordenada por inserção vazia com a capacidade inicial padrão (16) e o fator de carga (0,75).

`LinkedHashMap(int capacidade)`



Este construtor constrói um `LinkedHashMap` vazio com a capacidade inicial especificada.

`LinkedHashMap(int capacidade, float fR)`

Esse construtor constrói um `LinkedHashMap` vazio com a capacidade inicial especificada e o fator de carga.

`LinkedHashMap(Map m)`

Esse construtor constrói um `HashMap` vinculado ordenado por inserção com os mesmos mapeamentos que o mapa especificado.

`LinkedHashMap(int capacidade, float fR, boolean pedido)`

Esse construtor constrói uma instância `LinkedHashMap` vazia com a capacidade inicial especificada, o fator de carga e o modo de pedido.

Métodos importantes suportados pelo `LinkedHashMap`

Class `clear()`

Remove todos os mapeamentos do mapa.

`containsValue(object value)>`

Retorna `true` se este mapa mapeia uma ou mais chaves para o valor especificado.

`get(Object key)`

Retorna o valor para o qual a chave especificada está mapeada ou nula se este mapa não contiver mapeamento para a chave.

`removeEldestEntry(Map.Entry eldest)`



Retorna true se este mapa remover sua entrada mais antiga.

O seguinte programa demonstra o uso do LinkedHashMap:

```
package br.com.java.aula;
```

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

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

```
public class LinkedHashMapDemo {
```

```
    public static void main (String args[]){
```

```
        //Aqui o pedido de inserção mantém
```

```
        Map<Integer, String>lmap = new LinkedHashMap<Integer,  
String>();
```

```
        lmap.put(12, "Maria");
```

```
        lmap.put(5, "Nicole");
```

```
        lmap.put(23, "Samuel");
```

```
        lmap.put(9, "Sonia");
```

```
        System.out.println("LinkedHashMap antes da modificação" +  
lmap);
```

```
        System.out.println("Funcionario ID 12 exists: "  
+lmap.containsKey(12));
```

System.out.println("Funcionario André Exists:

" + lmap.containsValue("Andre"));



System.out.println("Número total de funcionários: " +
lmap.size());

System.out.println("Removendo funcionário com ID 5: " +
lmap.remove(5));

System.out.println("Removendo funcionário com ID 3 (o qual
não existe): " + lmap.remove(3));

System.out.println("LinkedHashMap depois da modificação"
+ lmap);

}

}

Resultado:

```
LinkedHashMap antes da modificação{12=Maria, 5=Nicole, 23
=Samuel, 9=Sonia}
Funcionario ID 12 exists: true
Funcionario André Exists: false
Número total de funcionários: 4
Removendo funcionário com ID 5: Nicole
Removendo funcionário com ID 3 (o qual não existe): null
LinkedHashMap depois da modificação{12=Maria, 23=Samuel,
9=Sonia}
```


Um `TreeMap` é um `Map` que mantém suas entradas em ordem crescente, classificadas de acordo com a ordem natural das chaves ou de acordo com um `Comparador` fornecido no momento do argumento do construtor `TreeMap`. A classe `TreeMap` é eficiente para percorrer as chaves em uma ordem classificada. As chaves podem ser classificadas usando a interface `Comparable` ou a interface `Comparator`. `SortedMap` é uma sub-interface de `Map`, que garante que as entradas no mapa sejam classificadas. Além disso, ele fornece os métodos `firstKey()` e `lastKey()` para retornar a primeira e a última chave no mapa, e `headMap(toKey)` e `tailMap(fromKey)` para retornar uma parte do mapa cujas chaves são menores que `toKey` e maiores que ou igual a `fromKey`.

Construtores `TreeMap`

`TreeMap()`

Construtor `TreeMap` padrão

`TreeMap(Map m)`

Isso é usado para criar o `TreeMap` com base na implementação de mapa existente `m`.

`TreeMap(SortedMap m)`

Isso é usado para criar o `TreeMap` com base na implementação de mapa existente `m`.

`TreeMap(Comparator())`

Isso é usado para criar o `TreeMap` com pedidos com base na saída do comparador.



```
import java.util.Map;

import java.util.TreeMap;

public class TreeMapDemo {

    public static void main(String[] args) {

        //Criando Map de frutas e os preços de cada uma

        Map<String, Integer> tMap = new TreeMap<String, Integer>

();

        tMap.put("Laranja", 12);

        tMap.put("Maçã", 25);

        tMap.put("Manga", 45);

        tMap.put("Caqui", 10);

        tMap.put("Banana", 4);

        tMap.put("Morango", 90);

        System.out.println("Ordenar frutas por nome: "+tMap);

        tMap.put("Uva", 87);

        tMap.remove("Caqui");

        System.out.println("Conjunto de frutas ordenadas por nome

atualizado: "+tMap);
```

}



}

Resultado:

```
Ordenar frutas por nome: {Banana=4, Caqui=
10, Laranja=12, Maçã=25, Manga=45, Morango=
90}
Conjunto de frutas ordenadas por nome
atualizado:{Banana=4, Laranja=12, Maçã=25,
Manga=45, Morango=90, Uva=87}
```

```
import java.util.*;
```

```
public class ContaPalavras {
```

```
    public static void main(String[] args) {
```

```
        // Ver o texto numa string
```

```
        String text = "Bom dia alunos. Boa aula. Bons estudos! ";
```

```
        // Crie um TreeMap para conter as palavras como chave e
        contar como valor
```

```
        TreeMap<String, Integer> map = new TreeMap<String,
        Integer>();
```

```
        String[] words = text.split(" "); //Divisão de sentenças com
        base em String
```

```
        for (int i = 0; i < words.length; i++) {
```

```
String key = words[i].toLowerCase();
```



```
if (key.length() > 0) {
```

```
    if (map.get(key) == null) {
```

```
        map.put(key, 1);
```

```
    } else {
```

```
        int value = map.get(key).intValue();
```

```
        value++;
```

```
        map.put(key, value);
```

```
    }
```

```
}
```

```
}
```

```
System.out.println(map);
```


```
}
```

```
}
```

Resultado:

```
{alunos= 1, bom=2, dia=1, estudo!=1}
```

Classe Collections de Java

A classe Collections consiste exclusivamente em métodos estáticos que operam ou retornam coleções. Ele contém algoritmos polimórf.  que operam em coleções, “wrappers”, que retornam uma nova coleção apoiada por uma coleção especificada,

Algum método útil na classe Collections:

Assinatura do método	Descrição
Collections.sort(List minhaLista)	Classifique o minhaLista (implementação de qualquer interface da Lista) fornecido um argumento na ordem natural.
Collections.sort(List, comparator c)	Classifique o minhaLista (implementação de qualquer interface da Lista) conforme a ordem do comparador c (a classe c deve implementar a interface do comparador)
Collections.shuffle(List minhaLista)	Coloca os elementos de minhaLista ((implementação de qualquer interface de Lista) em ordem aleatória
Collections.reverse(List minhaLista)	Inverte os elementos de minhaLista ((implementação de qualquer interface de lista)
Collections.binarySearch(List mlist, T key)	Pesquisa o mlist (implementação de qualquer interface da Lista) pelo objeto especificado usando o algoritmo de pesquisa binária.
Collections.copy(List dest, List src)	Copie a lista de origem na lista de destino.
Collections.frequency(Collection c, Object o)	Retorna o número de elementos na classe de coleção especificada c (que implementa a interface Collection pode ser List, Set ou Queue) igual ao objeto especificado
Collections.synchronizedCollection(Collection c)	Retorna uma coleção sincronizada (segura para thread) apoiada pela coleção especificada.

Vamos dar o exemplo de classificação de lista usando a classe Collections. Podemos classificar qualquer coleção usando a classe de utilitário “Coleções”. ie; ArrayList of Strings pode ser classificado em ordem alfabética usando esta classe de utilitário. A própria classe ArrayList não está fornecendo nenhum método para classificar. Usamos métodos estáticos da classe Collections para fazer isso. O programa abaixo mostra o uso dos métodos reverse (), shuffle (), frequency () também.

```
package br.com.java.aula;

import java.util.Collections;

import java.util.ArrayList;

import java.util.List;

public class ColecoesDemo {

    public static void main(String[] args) {

        List<String>student<String>List = new ArrayList();

        alunosLista.add("Naomi");

        alunosLista.add("Maria");

        alunosLista.add("Amanda");

        alunosLista.add("Paulo");

        alunosLista.add("Sofia");

        alunosLista.add("Naomi");
```

```
alunosLista.add("Zelia");
```



```
System.out.println("Lista original " + alunosLista);
```

```
Collections.sort(alunosLista);
```

```
System.out.println("Lista ordenada alfabeticamente " +  
alunosLista);
```

```
Collections.reverse(alunosLista);
```

```
System.out.println("Lista invertida " + alunosLista);
```

```
Collections.shuffle(alunosLista);
```

```
System.out.println("Lista aleatória " + alunosLista);
```

```
System.out.println("Verificando ocorrências de Naomi: "
```

```
+ Collections.frequency(alunosLista,  
"Naomi"));
```

```
}
```

```
}
```

Resultado:

```

Lista original [Naomi, Amanda, Maria, Paula, Sofia, Naomi, Zelia]
Lista ordenada alfabeticamente [Amanda, Maria, Naomi, Naomi, Paula, Sofia, Zelia]
Lista invertida [Zelia, Sofia, Paula, Naomi, Naomi, Maria, Amanda]
Lista aleatória [Sofia, Naomi, Zelia, Maria, Amanda, Naomi, Paula]
Verificando ocorrências de Naomi: 2

```

Usando a classe Collections, podemos copiar um tipo de coleção para outro tipo. As coleções nos fornecem o método de cópia para copiar todos os elementos que dão origem ao destino. O programa abaixo demonstra o uso da função de cópia. Aqui, o tamanho da coleção de origem e destino deve ser o mesmo, caso contrário, obteremos a exceção a seguir.

```

Exception in thread "main" java.lang.IndexOutOfBoundsException: Source does not fit in dest
at java.util.Collections.copy(Unknown Source)
at utility.ArrayToListDemo.main(ArrayToListDemo.java:26)

```

```
import java.util.Collections;
```

```
import java.util.*;
```

```
public class CopiaListaDemo {
```

```
    public static void main(String[] args) {
```

```
        List <Integer>minhaPrimeiraLista = new ArrayList<Integer>
();
```

```
        List <Integer> minhaSegundaLista = new ArrayList<Integer>
();
```

```
        minhaPrimeiraLista.add(10);
```

```
        minhaPrimeiraLista.add(20);
```

```
        minhaPrimeiraLista.add(20);
```



```
minhaPrimeiraLista.add(50);
```



```
minhaPrimeiraLista.add(70);
```

```
minhaSegundaLista.add(11);
```

```
minhaSegundaLista.add(120);
```

```
minhaSegundaLista.add(120);
```

```
minhaSegundaLista.add(150);
```

```
minhaSegundaLista.add(170);
```

```
System.out.println("Primeira lista-" + minhaPrimeiraLista);
```

```
System.out.println("Segunda lista-" + minhaSegundaLista);
```

```
Collections.copy(minhaSegundaLista, minhaPrimeiraLista );
```

```
        System.out.println("Segunda lista depois da cópia-" +  
minhaSegundaLista);
```

```
    }
```

```
}
```

Resultado:

```
Primeira lista-[10, 20, 20, 50, 70]  
Segunda lista-[11, 120, 150, 170] ⓘ  
Segunda lista depois da cópia-[10, 20,  
20, 50, 70]
```

Atividade extra

Vídeo: “Ordenação com Collection”

Link: <https://www.youtube.com/watch?v=e8zWALRr0Ro>

Referências Bibliográficas

BARNES, D. J. KOLLING, M. **Programação orientada a objetos com java: uma introdução prática usando o bluej**. 4.ed. Pearson: 2009.

FELIX, R. (Org.). **Programação orientada a objetos**. Pearson: 2017.

MEDEIROS, L. F. de. **Banco de dados: princípios e prática**. Intersaberes: 2013;

ORACLE. Java Documentation, 2021. **Documentação oficial da plataforma Java**. Disponível em: < <https://docs.oracle.com/en/java/> >.

Ir para exercício