PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
DEPARTAMENTO DE COMPUTAÇÃO
TÉCNICAS DE PROGRAMAÇÃO 1 – CMP 1046
PROF. MSC. ANIBAL SANTOS JUKEMURA

[Collections]

PUC GOIÁS

Agenda:

- Introdução
- Características
- Exemplos

Collections

- Desde as primeiras versões, Java dispõe das estruturas de **arrays** e as classes **Vector** e **Hashtable**.

- No entanto, além da dificuldade em implementar estruturas de dados utilizando arrays, **os desenvolvedores sentiam falta de classes que implementassem estruturas como listas ligadas e tabelas de espalhamento** (**hash**).

- Para atender a essas necessidades, **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

- O uso das interfaces disponíveis no Collections Framework evita que o programador desperdice esforço para desenvolver suas próprias estruturas e se concentre em outras partes importantes da programação;

- Suas estruturas de dados e algoritmos de alta qualidade e excelente desempenho melhoram a qualidade e a performance das aplicações; reduz o esforço de ter que aprender e usar novas APIs;

- Promove o reuso de software e permite interoperabilidade entre APIs não relacionadas.

Collections

- **Collections Framework** é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade, que pode ser chamada coleção, ou collection. A Collections Framework contém os seguintes elementos:

  - **Interfaces**: tipos abstratos que representam as coleções. Permitem que coleções sejam manipuladas tendo como base o conceito "Programar para interfaces e não para implementações", desde que o acesso aos objetos se restrinja apenas ao uso de métodos definidos nas interfaces;

  - **Implementações**: são as implementações concretas das interfaces;

  - **Algoritmos**: são os métodos que realizam as operações sobre os objetos das coleções, tais como busca e ordenação.

PUC GOIÁS

Collections

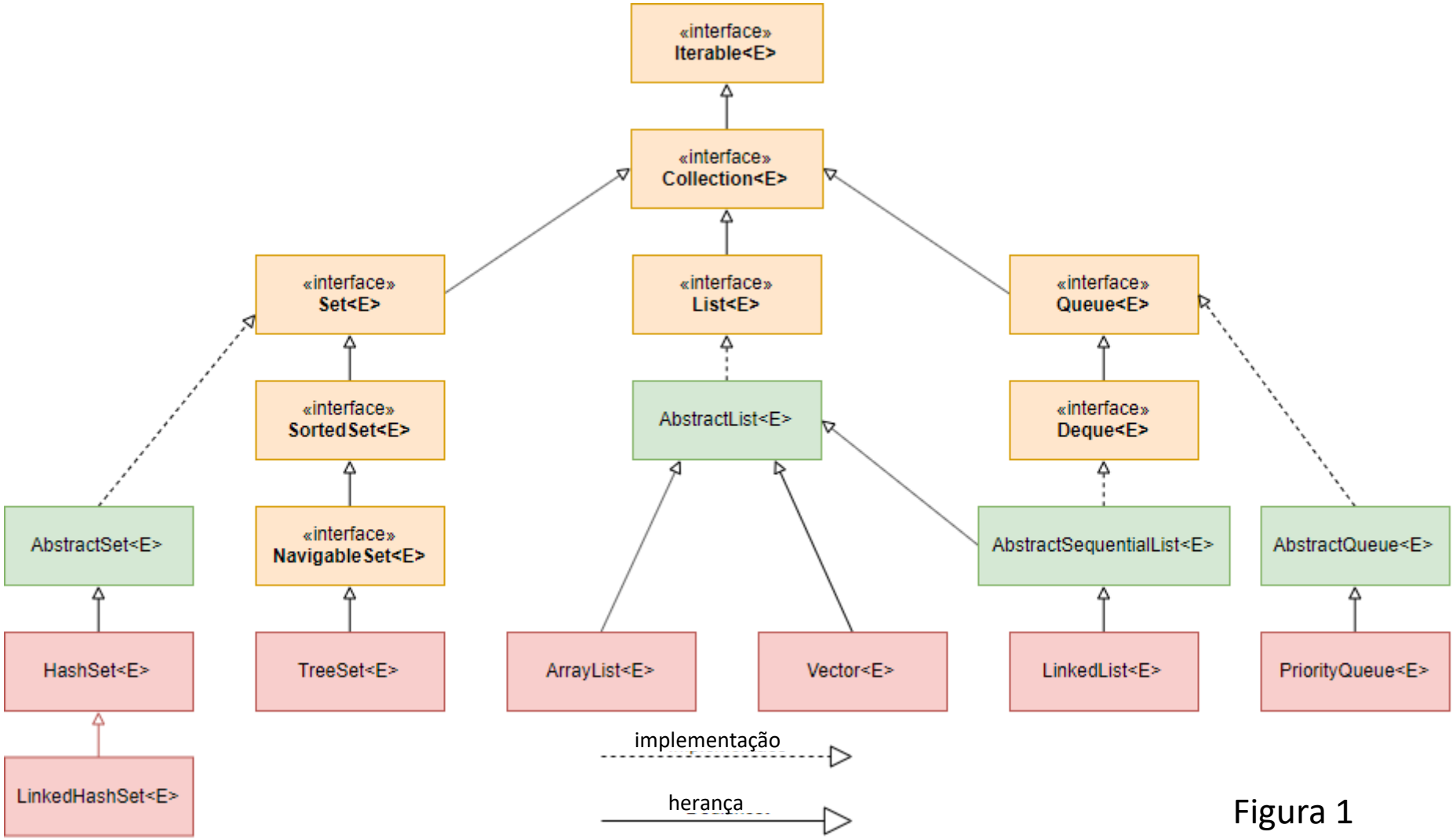


Figura 1

PUC GOIÁS

Collections

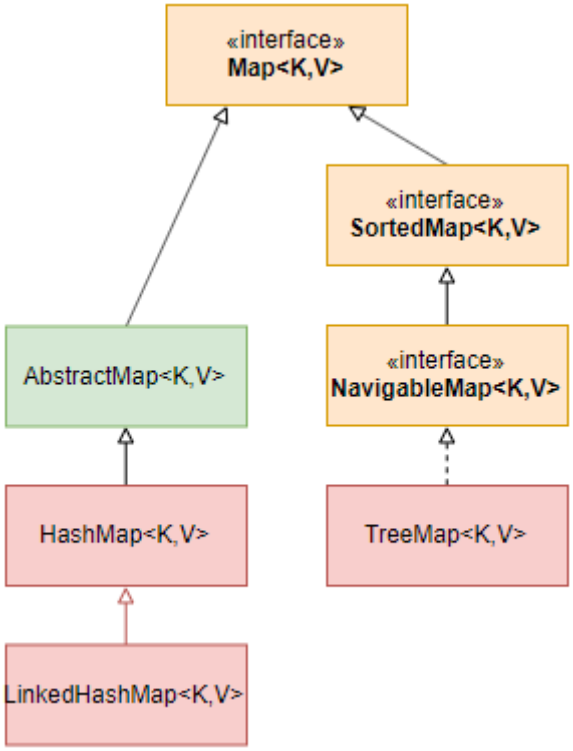


Figura 2

A hierarquia da Collections Framework tem uma segunda árvore. São as classes e interfaces relacionadas a mapas, que não são derivadas de Collection, como mostra a Figura 2. Essas interfaces, mesmo não sendo consideradas coleções, podem ser manipuladas como tal.

PUC GOIÁS

Collections

- **Collection** – está no topo da hierarquia. Não existe implementação direta dessa interface, mas ela define as operações básicas para as coleções, como adicionar, remover, esvaziar, etc.;

- **Set** – interface que define uma coleção que não permite elementos duplicados. A interface **SortedSet**, que estende Set, possibilita a classificação natural dos elementos, tal como a ordem alfabética;

- **List** – define uma coleção ordenada, podendo conter elementos duplicados. Em geral, o usuário tem controle total sobre a posição onde cada elemento é inserido e pode recuperá-los através de seus índices. Prefira esta interface quando precisar de acesso aleatório, através do índice do elemento;

PUC GOIÁS

Collections

- **Queue** – um tipo de coleção para manter uma lista de prioridades, onde a ordem dos seus elementos, definida pela implementação de **Comparable** ou **Comparator**, determina essa prioridade. Com a interface fila pode-se criar **filas** e **pilhas**;

- **Map** – mapeia chaves para valores. Cada elemento tem na verdade dois objetos: uma chave e um valor. Valores podem ser duplicados, mas chaves não. **SortedMap** é uma interface que estende Map, e permite classificação ascendente das chaves. Uma aplicação dessa interface é a classe **Properties**, que é usada para persistir propriedades/configurações de um sistema, por exemplo.

PUC GOIÁS

Collections - implementações

Implementações					
Interfaces	**Tabela de Espalhamento**	**Array Redimensionável**	**Árvore**	**Lista Ligada**	**Tabela de Espalhamento + Lista Ligada**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Tabela 1. Implementações de uso geral

PUC GOIÁS

Collections

- **Nota**: Não confunda a **interface Collection** com **a classe Collections**. Essa classe oferece métodos estáticos utilitários que podem manipular coleções. Outra classe utilitária é Arrays, cujos métodos estáticos são aplicados a arrays.

PUC GOIÁS

Collections - Algoritmos

Método	Descrição
`sort`	Classifica os elementos de uma `List`.
`binarySearch`	Localiza um objeto em uma `List` usando o algoritmo de pesquisa binária de alto desempenho introduzido na Seção 7.15 e discutido em detalhes na Seção 19.4.
`reverse`	Inverte os elementos de uma `List`.
`shuffle`	Ordena aleatoriamente os elementos de uma `List`.
`fill`	Configura todo elemento `List` para referir-se a um objeto especificado.
`copy`	Copia referências de uma `List` em outra.
`min`	Retorna o menor elemento em uma `Collection`.
`max`	Retorna o maior elemento em uma `Collection`.
`addAll`	Acrescenta todos os elementos em um arrav a uma `Collection`.
`frequency`	Calcula quantos elementos da coleção são iguais ao elemento especificado.
`disjoint`	Determina se duas coleções não têm nenhum elemento em comum.

Listas

Na lista de interfaces da estrutura de coleções destacam-se os conjuntos, listas, filas e mapas. Vamos começar pelas listas:

Lista (List):

- É uma coleção indexada de objetos às vezes chamada de sequência.
- Como nos vetores, índices de List são baseados em zero, isto é, o índice do primeiro elemento é zero.
- Além dos métodos herdados de Collection, List fornece métodos para manipular elementos baseado na sua posição (ou índice) numérica na lista, remover determinado elemento, procurar as ocorrências de um dado elemento e percorrer sequencialmente (ListIterator) todos os elementos da lista.
- A interface List é implementada por várias classe, incluídas as classes **ArrayList** (implementada como vetor), **LinkedList** e **Vector** (obsoleto).

Listas

ArrayList

PUC GOIÁS

- Usaremos esta classe na implementação de vetores dinâmicos (ou redimensionáveis).
- Principais métodos:

- boolean **add**(**Object** element): Adiciona o elemento especificado no final da lista.

- void **add**(**int** index, **Object** element): Insere o elemento especificado na posição indicada da lista.

- void **clear**(): Remove todos os elementos da lista.

- boolean **contains**(**Object** element): Retorna verdadeiro se a lista contém o elemento especificado e falso caso contrário.

- Object **get**(**int** index): Retorna o i-ésimo elemento da lista.

- int **indexOf**(**Object** element): Retorna a posição da primeira ocorrência do elemento especificado na lista.

Listas

ArrayList

- Usaremos esta classe na implementação de vetores dinâmicos (ou redimensionáveis).
- Principais métodos:
- boolean **isEmpty**(): Retorna verdadeiro se a lista estiver vazia e falso caso contrário.
- int **lastIndexOf**(Object element): Retorna a posição da última ocorrência do elemento especificado na lista.
- Object **remove**(int index): Remove o i-ésimo elemento da lista.
- Object **set**(int index, Object element): Substitui o i-ésimo elemento da lista pelo elemento especificado.
- int **size**(): Retorna o número de elementos da lista.

Listas

ArrayList: exemplo 1 – Lista de Inteiros

Neste exemplo em sala realizaremos as seguintes operações:

1 – Criar uma lista de Inteiros:

```
ArrayList<Integer> numeros = new ArrayList();
```

2 - Usar o método add() para gravar números digitados pelo usuário:

```
numeros.add(ler.nextInt());
```

3 - Mostrar os "n" números da lista (usando o índice) e o método método size() que retorna o número de elementos da lista (tamanho da lista):

```
int n = numeros.size();
numeros.get(i);
```

4- Vamos remover o i-ésimo número da lista:

```
numeros.remove(i);
```

5- Mostar os "n" elementos da lista (usando for-each)

```
for (int num: numeros)
```

Listas

ArrayList: exemplo 2 - Agenda

Neste exemplo em sala realizaremos as seguintes operações:

1 – Criar uma lista de Strings:

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

2 - Usar o método add() para gravar contatos na agenda:

```
agenda.add("Darth Vader;62 9999-1111");
```

3 - Mostrar os "n" contatos da agenda (usando o índice) e o método método size() que retorna o número de elementos da agenda (tamanho da lista):

```
int n = agenda.size();
agenda.get(i);
```

4- Vamos remover o i-ésimo contato da agenda:

```
agenda.remove(i);
```

5- Mostar os "n" contatos da agenda (usando for-each)

```
for (String contato: agenda)
```

6- mostrando os "n" contatos da agenda (com iterator)

```
Iterator<String> iterator = agenda.iterator();
iterator.hasNext()
iterator.next()
```

Listas

LinkedList

- Declaração:
  - **import** java.util.LinkedList;
  - **Import** java.util.List;
  - **List**<TIPO> fila = new **LinkedList**();

- Principais métodos:

- boolean **add**(**Object** element): Adiciona o elemento especificado no final da lista.

- boolean **add**(int i, **Object** element): Adiciona o elemento especificado na posição i da lista.

- boolean **contains**(**Object** element): Retorna verdadeiro se a fila contém o elemento especificado e falso caso contrário.

- void **clear**(): Remove todos os elementos da fila.

- Object **get**(**int** index): Retorna o i-ésimo elemento da lista.

Listas

LinkedList

- Principais métodos:

- boolean **isEmpty**( ): Retorna true se a fila estiver vazia. Retorna false, caso contrário.
- Object **remove**( ): Remove o primeiro elemento da fila. Executa um throw exception do tipo **NoSuchElementException** se a fila estiver vazia.
- int **size**(): Retorna o número de elementos da fila.
- Object **set**(int index, Object element): Substitui o i-ésimo elemento da lista pelo elemento especificado.
- Iterator<E> **iterator**(): retorna um iterador para controle da colação.

PUC GOIÁS

Fila (*Queue*)

Uma coleção utilizada para manter uma "fila" de elementos. Existe uma ordem linear para as filas que é a "ordem de chegada". As filas devem ser utilizadas quando os itens deverão ser processados de acordo com a ordem "PRIMEIRO-QUE-CHEGA, PRIMEIRO-ATENDIDO". Por esta razão as filas são chamadas de Listas **FIFO**, termo formado a partir de "*First-In, First-Out*".

PUC GOIÁS

Fila (*Queue*)

- Fila é uma estrutura de dados linear “especial”, que permite apenas a eliminação de elementos no início e no fim realiza-se a operação de inserção.

- Uma das formas mais simples de se implementar uma fila em java é instanciá-la como uma lista simples encadeada, ou LinkedList.

- A classe LinkedList implementa a interface de fila, para que possamos determinar qual tipo de fila podemos usar.

Filas

Queue

- Declaração:
  - **import** java.util.Queue;
  - **import** java.util.LinkedList;
  - **Queue**<TIPO> fila = new **LinkedList**();

- Principais métodos:

- void **offer**(**Object** element): Insere um elemento no final da fila sem que ocorra uma violação nas restrições de capacidade da fila. A diferença com o método add ocorre quando a fila possui tais restrições, sendo que o método add somente irá inserir o elemento, gerando uma exceção, caso uma das restrições seja violada.

- boolean **add**(**Object** element): Adiciona o elemento especificado no final da fila.

- boolean **contains**(**Object** element): Retorna verdadeiro se a fila contém o elemento especificado e falso caso contrário.

- Object **element**( ): Retorna primeiro elemento da fila sem removê-lo. Executa um throw exception do tipo **NoSuchElementException** se a fila estiver vazia.

Filas

Queue

PUC GOIÁS

- Principais métodos:
- Object **peek**( ): Retorna primeiro elemento da fila sem removê-lo. Retorna **null** se a fila estiver vazia.
- boolean **isEmpty**( ): Retorna true se a fila estiver vazia. Retorna false, caso contrário.
- Object **remove**( ): Remove o primeiro elemento da fila. Executa um throw exception do tipo **NoSuchElementException** se a fila estiver vazia.
- Object **poll**( ): Remove o primeiro elemento da fila. Retorna **null** se a fila estiver vazia.
- int **size**(): Retorna o número de elementos da fila.
- Iterator<E> **iterator**(): retorna um iterador para controle da colação.
- void **clear**(): Remove todos os elementos da fila.

PriorityQueue

- **PriorityQueue**, que implementa a interface Queue, ordena elementos por sua ordem natural como especificado pelo método compareTo dos elementos Comparable ou por um objeto Comparator que é fornecido pelo construtor.

- A classe **PriorityQueue** fornece funcionalidades que permitem inserções na ordem de classificação na estrutura de dados subjacente e exclusões a partir da frente da estrutura de dados subjacente.

- Ao adicionar elementos a uma **PriorityQueue**, os elementos são inseridos na ordem de prioridade de tal modo que o elemento de maior prioridade (isto é, o maior valor) será o primeiro elemento removido da **PriorityQueue**.

Filas

Queue: exemplo 1 – Fila de Inteiros

Neste exemplo em sala realizaremos as seguintes operações:

1 – Criar uma fila de Inteiros:

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

2 - Usar o método add() ou offer() para gravar números digitados pelo usuário:

```
fila.add(ler.nextInt());
fila.offer(ler.nextInt());
```

3- Vamos remover um elemento da fila:

```
fila.remove();
fila.poll();
```

5- Mostar os "n" elementos da fila (usando for-each)

```
for (int num: fila)
```

6- mostrando os "n" elementos da fila (com iterator)

```
Iterator<Integer> iterator = fila.iterator();
iterator.hasNext()
iterator.next()
```

Vejamos um exemplo em Sala

PILHA (STACK)

PUC GOIÁS

Implementações:

- API JAVA
- Estruturas estáticas: vetores
- Estruturas dinâmicas: listas simplesmente encadeadas

PILHA (STACK)

- Declaração:
  - **import** java.util.Stack;
  - **Stack<TIPO>** pilha = new **Stack**();

- Principais métodos:
  - boolean empty(): testa se a pilha está vazia.
  - E peek(): retorna o objeto que está no topo da pilha sem removê-lo.
  - E pop(): remove (desempilha) o objeto que está no top da pilha.
  - push (E item): empilha o objeto E na pilha.
  - int search(Object O): retorna a posição de um objeto que está na pilha.

PUC GOIÁS

PILHA (STACK)

- Métodos herdados da classe Vector
  - clear(): esvazia a pilha.
  - boolean contains(Object O): verifica se o objeto está na pilha.
  - Iterator <E> iterator(): iterator usado para retornar os elementos da pilha (da base ao topo).
  - E firstElement (): retorna o primeiro elemento da pilha sem removê-lo.
  - int size(): retorna o tamanho da pilha.

(Cuidado): esses demais métodos foram herdados da classe Vector e descaracterizam o funcionamento de uma pilha, apesar de poderem ser utilizados.

add, addAll, addAll, addElement, capacity, clone, containsAll, elementAt, elements, ensureCapacity, equals, get, indexOf, insertElementAt, lastElement, lastIndexOf, listIterator, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, subList.

INTERFACE MAP

PUC GOIÁS

- Essa interface é um objeto que mapeia **valores** para **chaves**, ou seja, através da chave consegue ser acessado o valor configurado
- A chave não pode ser repetida ao contrário do valor, mas se caso tiver uma chave repetida é sobrescrito pela última chamada.
- Faz parte do pacote **java.util** e não possui métodos da interface Collection.

- Algoritmo hash (*hash code*): Esse algoritmo transforma uma grande quantidade de dados em uma pequena quantidade de informações, sendo que o mecanismo de busca se baseia na **construção de índices**.

INTERFACE MAP

- Um exemplo prático pode ser usado como uma lista telefônica onde a letra seria o índice a ser procurado, para conseguir achar mais fácil o nome desejado.

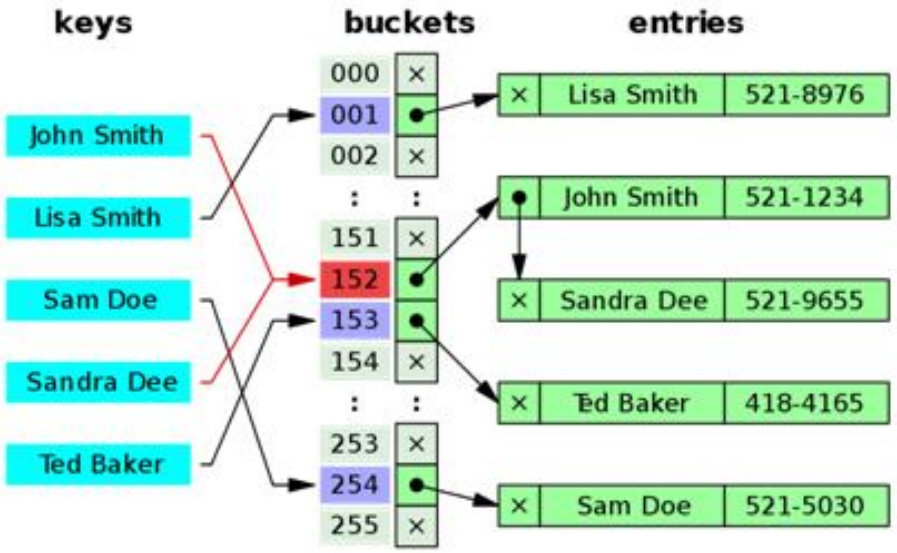


Figura 1. Exemplo do mecanismo do algoritmo hash

INTERFACE MAP

Sintaxe: Map <E> mapa = new Type();

- **E** – conjunto chave e valor mapeado <K, V>
- **Type** - é o tipo de objeto da coleção a ser usado (HashMap, TreeMap, HashTable ...)

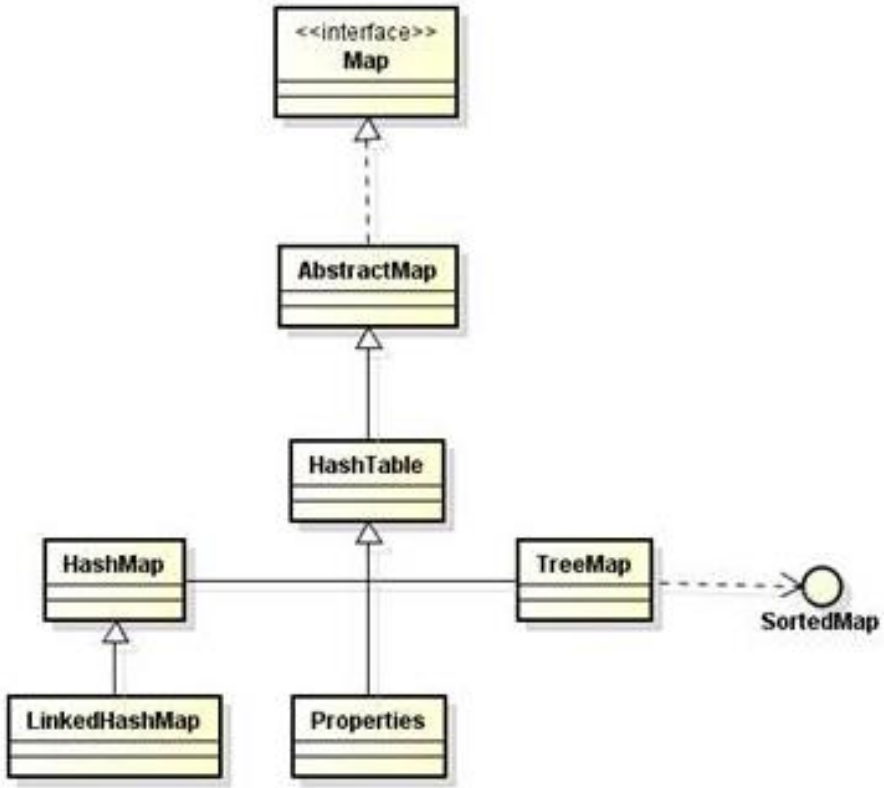


Figura 2. Hierarquia da Interface Map

INTERFACE MAP

PUC GOIÁS

Principais métodos:

clear()	remove todos os elementos do mapa
containsKey()	returna verdadeiro se o mapa contém a chave requisitada
containsValue()	returna verdadeiro se o mapa contém o valor requisitado
equals()	compara um objeto com o mapa
get()	retorna o valor requisitado pela chave
keySet()	retorna um conjunto que contém todas as chaves do mapa
put()	adiciona o par chave-valor no mapa
remove()	remove o par chave-valor do mapa
size()	retorna o número de pares chave-valor contidos no mapa
isEmpty()	retorna o código de hash para o mapa

INTERFACE MAP

PUC GOIÁS

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

public class TestaInterfaceMap {

  public static void main(String[ ] args) {

    Map<Integer, String> mapaNomes = new HashMap<Integer, String>();
    mapaNomes.put(1, "João Delfino");
    mapaNomes.put(2, "Maria do Carmo");
    mapaNomes.put(3, "Claudinei Silva");

    System.out.println(mapaNomes);

    //resgatando o nome da posição 2
    System.out.println(mapaNomes.get(2));

  }
}
```

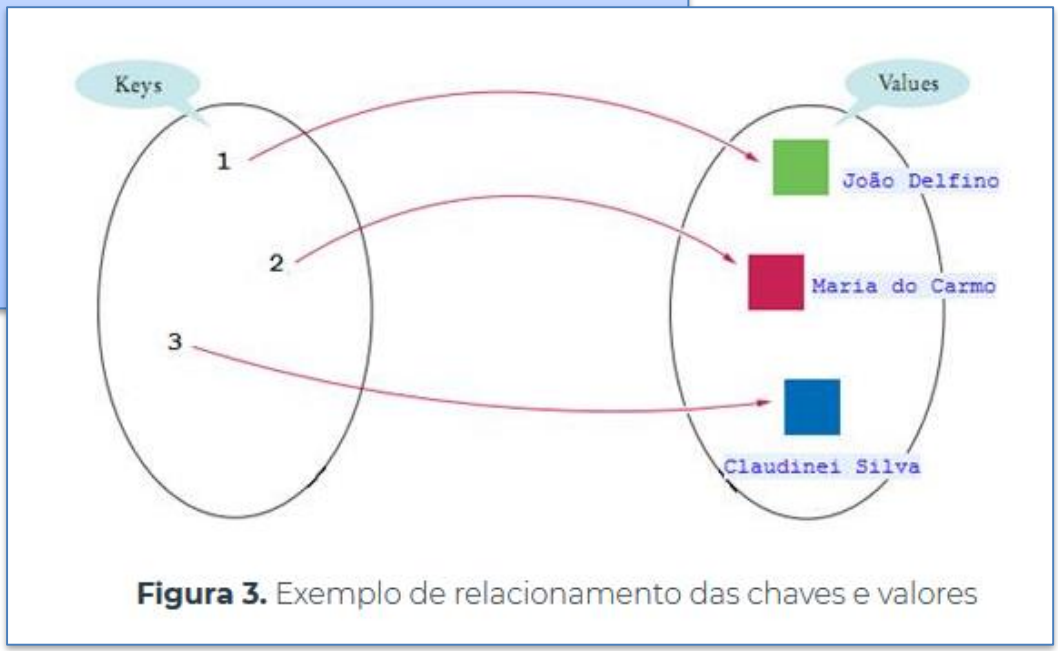


Figura 3. Exemplo de relacionamento das chaves e valores

INTERFACE MAP

Principais implementações:

Classe HashMap

- Essa classe é a implementação da interface Map mais trabalhada no campo de desenvolvimento.

- Características

  - Os elementos não são ordenados;
  - É rápida na busca/inserção de dados;
  - **Permite inserir valores e chaves nulas**;

Sintaxe:

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

Map<K,V> mapa = new HashMap<K,V>()
```

INTERFACE MAP

Principais implementações:

Classe HashTable

- Essa classe trabalha com algoritmo hash para conversão das chaves e um mecanismo de pesquisa de valores, sendo que tem seus métodos sincronizados (thread-safe) que permitem checar acessos concorrentes e armazenagem. Também possui uma eficiente pesquisa de elementos baseados em chave-valor, mas **não aceita nem chaves e nem valores nulos**.

Sintaxe:

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

Map<K,V> mapa = new HashTable<K,V>()
```

INTERFACE MAP

Principais implementações:

Classe TreeMap

- Árvore do tipo “red black” que é **ordenada conforme a ordem natural de suas chaves**, ou através de um comparador quando definida em tempo de “criação”.

Sintaxe:

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

Map<K,V> mapa = new TreeMap<K,V>()
```

INTERFACE MAP

Interação com a interface MAP

Para interagir sobre um mapa é preciso trabalhar com a interface **Collection ou métodos set()** para converter esse mapa em um conjunto de dados.

```
for (String chave: veiculos.keySet())
  System.out.println (chave + " - " + veiculos.get(chave));
```

≠

```
for(Integer valor: veiculos.values())
   System.out.println (valor + " - " + veiculos.get(valor));
```

INTERFACE MAP

PUC GOIÁS

Interação com a interface MAP

Para interagir sobre um mapa é preciso trabalhar com a interface **Collection ou métodos set()** para converter esse mapa em um conjunto de dados.

```
import java.util.Collection;
import java.util.Iterator;
import java.util.Set;
import java.util.Map.Entry;
```

INTERFACE MAP

PUC GOIÁS

Interface MAP com ArrayLists

Até o momento foi visto como utilizar as implementações de MAPS. Entretanto, obeservou-se que, para valores únicos, quando é utilizada a mesma chave, o valor atual é sobreposto.

Veja exemplo:

```
Map<String, Integer> veiculos = new Hashtable();

veiculos.put("BMW", 5);
veiculos.put("Mercedes", 3);
veiculos.put("Audi", 4);
veiculos.put("Ford", 10);
veiculos.put("Fiat", 7);
veiculos.put("Audi", 40);
```

Rersultado de veiculos.get(busca) = 40

INTERFACE MAP

PUC GOIÁS

Interface MAP com ArrayLists

Implementar uma Hash Map com ArrayLists, torna a estrutura mais parecida com a TAD (Tipo Abstrato de Dados) Hash Table:

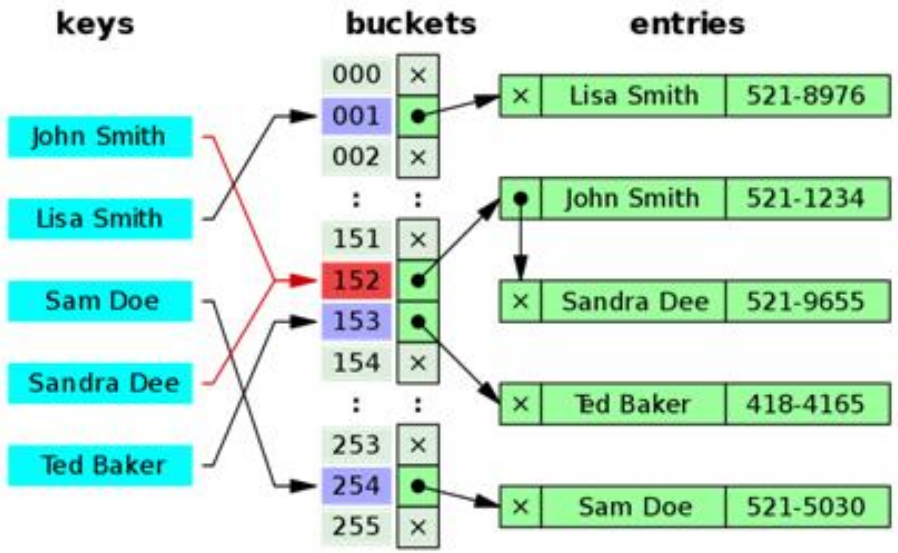


Figura 1. Exemplo do mecanismo do algoritmo hash

INTERFACE MAP

PUC GOIÁS

Interface MAP com ArrayLists - Implementação

Sigam os passos:

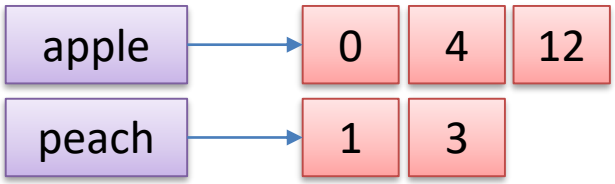
Declaração: treemap, hashmap ou hastable

- TreeMap< **Chave**, ArrayList< **Tipo** >> map = new TreeMap< **Chave**,ArrayList< **Tipo** >>();
- HashMap<**Chave**,ArrayList<**Tipo**>> map=new HashMap< **Chave**,ArrayList<**Tipo**>>();
- Map<Str **Chave** ing, ArrayList< **Tipo** >> map = new Hashtable< **Chave**,ArrayList< **Tipo** >>();

Onde: **Chave** é qualquer tipo de dados simples para ser usado como indexador (**Chave**)

Exemplo: Chave = String e ArrayList de Inteiros:
HashMap<**String**,ArrayList<**Integer**>> map=new HashMap< **String**,ArrayList< **Integer** >>();

map.put("apple", new ArrayList<Integer>(Arrays.asList(0, 4, 12)));
map.put("peach", new ArrayList<Integer>(Arrays.asList(1,3)));



INTERFACE MAP

Interface MAP com ArrayLists - Implementação

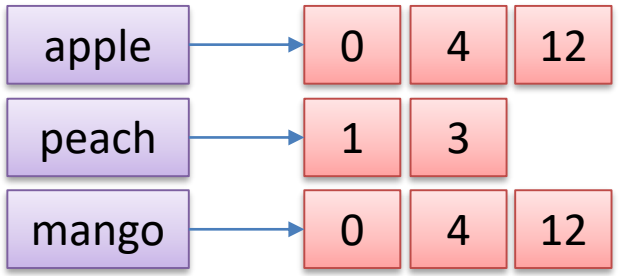
Para inserir uma lista fora da declaração, basta criar a lista e inseri-la conforme exemplo anterior. Veja:

Exemplo: Chave = String e ArrayList de Inteiros:

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

map.put("apple", new ArrayList<Integer>(Arrays.asList(0, 4, 12)));
map.put("peach", new ArrayList<Integer>(Arrays.asList(1,3)));

 ArrayList<Integer> lista = new ArrayList();
    lista.add(11);
    lista.add(12);
    lista.add(13);
    map.put("mango", lista);
```



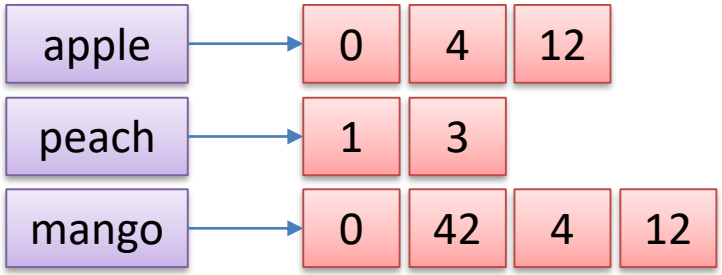
INTERFACE MAP

Interface MAP com ArrayLists - Implementação

Para atualizarmos o ArrayList na chave que escolhermos, basta chamar o método add o ArrayList. Veja o exemplo:

```
String key = "mango";
int number = 42;
map.get(key).add(1,number);
```

Insere na chave “mango” o número 42 na posição 1 do ArrayList



INTERFACE MAP

Interface MAP com ArrayLists - Implementação

Para imprimir o Hash Map, basta usar o Iterator como foi visto anteriormente, ou um Entry, com um ArrayList, através dos métodos getKey e getValue. Veja o exemplo:

```
for (Entry<String, ArrayList<Integer>> ee : map.entrySet())
{
   String key = ee.getKey();
   ArrayList<Integer> values = ee.getValue();
   System.out.println(key + values);
}
```

Ou simplesmente:

```
for (Entry<String, ArrayList<Integer>> ee2 : map.entrySet())
      System.out.println(ee2.getKey() + ee2.getValue());
```

PUC GOIÁS

BOAS PRÁTICAS DE PROGRAMAÇÃO

Observação de engenharia de software 16.1

`Collection` é comumente utilizada como um tipo de parâmetro nos métodos para permitir processamento polimórfico de todos os objetos que implementam a interface `Collection`.

Observação de engenharia de software 16.2

A maioria das implementações de coleção fornece um construtor que aceita um argumento `Collection`, permitindo, assim, que uma nova coleção a ser construída contenha os elementos da coleção especificada.

Dica de desempenho 16.1

`ArrayList`s comportam-se como `Vector`s sem sincronização e, portanto, executam mais rápido que `Vector`s, porque `ArrayList`s não têm o overhead de sincronização de thread.

Observação de engenharia de software 16.3

`LinkedList`s podem ser utilizadas para criar pilhas, filas e deques (double-ended queues — filas com dupla terminação). A estrutura de coleções fornece implementações de algumas dessas estruturas de dados.

Referência Bibliográfica Principal

• DEVMEDIA. Disponível em https://www.devmedia.com.br. Acessado em Julho de 2019.

• DEITEL, Harvey M. Java: Como Programar – 10 ed. Cap 16. 2015.