Documentação de COLLECTIONS - by Joaquim Guilherme

Collections Framework

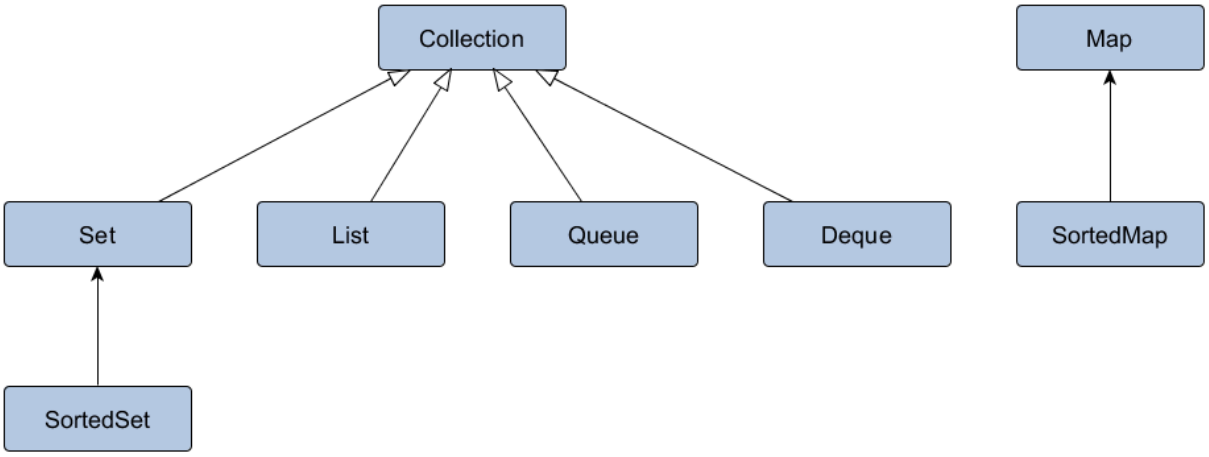
Introduzido a partir do Java 1.2 , o Collections Framework é um conjunto de interfaces e classes, que permite a manipulação de coleções de dados de forma muito mais eficiente e flexível.

Ele faz parte do pacote **java.util,** responsável pela **Collections API**, e foi desenvolvido com o objetivo de suprir as limitações na utilização de arrays, como tamanho fixo e ausência de métodos.

Entre suas principais classes estão:

- List
- Set
- Map
- Queue
- Deque

Hierarquia do Collections Framework



List

A **interface** List (Lista) é responsável por armazenar coleções de elementos de uma classe única em forma de lista, isto é, os elementos são armazenados por ordem de inserção e podem ser duplicados. Sua aplicação mais comum e a classe **ArrayList**

Principais Características

- Armazena elementos de maneira ordenada e sequencial
- Permite que elementos sejam duplicados (iguais)
- Permite o acesso a elementos por atraves de indices
- Não possui tamanho fixo

Métodos de List

1- **for-each**

O laço for-each é o mais comum para percorrer listas sem a preocupação com índices ou operações complexas, tornando o código muito mais simples e legível.

Ele apresenta três implementações:

- for (Pessoa pessoa : pessoas) { // ação }

Ou, a opção mais recente e com utilização de um método já estabelecido forEach e com a utilização de uma expressão lambda

- pessoas.forEach(pessoa -> // ação);

Ou até mais reduzida ainda com Method Reference

- pessoas.forEach(System.out::println);

2- **size()**

Retorna o número de elementos já adicionados à lista

- pessoas.size();

3- **get(index)**

Retorna o objeto correspondente ao endereço passado por parâmetro

- pessoas.get(0); // retorna o primeiro objeto

4- **add(object)**

Adiciona o elemento passado por parâmetro na lista de elementos

- pessoas.add(pessoa);

5- **remove(object)**

Remove o elemento (já existente na lista) passado por parâmetro da lista de elementos

- pessoas.remove(pessoa);

6- **clear()**

Exclui todos os elementos adicionados à lista

- pessoas.clear();

7- **isEmpty()**

Este é um método booleano que verifica se a lista está vazia

- pessoas.isEmpty();

Principais aplicações

1- List<T> t = new **ArrayList<>()**;

- array dinâmico
- sequencial
- permite elementos duplicados
- não possui limite fixo

2- List<T> t = new **LinkedList<>()**;

- array encadeado, no qual cada elemento contém uma referência ao próxi
- sequencial
- permite elementos duplicados
- não possui limite fixo

Map

A **interface** Map (Mapa) permite a criação de uma coleção de dados mapeada por elementos **chave-valor** (key-value) e pelo algoritmo **Hash Code**

Amplamente utilizada em grandes coleções de dados, é caracterizada pela sua alta eficiência para busca, atualização e recuperação de elementos

Sua principal implementação é a classe **HashMap**

Principais Características

- Mapeia chaves para valores
- Não possui os métodos da classe Collections
- Coleção mais eficiente para busca de elementos
- Não é ordenado
- Não permite elementos duplicados

Métodos de Map

1- **put(key,value)**

Permite adicionar um novo elemento a coleção

- pessoas.put(1,"Joaquim");

2- **get(key)**

Permite buscar um elemento a partir de sua key

- pessoas.get(1); // Joaquim

3- **remove(key)**

Remove um elemento da coleção a partir de sua key

- pessoas.remove(1); // remove Joaquim

4- **values()**

Retorna todos os valores atribuídos a uma key da coleção

- products.values();

5- **clear()**

Remove todos os elementos da coleção

- products.clear();

6- **isEmpty()**

Método booleano que retorna true se a coleção estiver vazia

- products.isEmpty();

7- **size()**

Retorna o numero int de elementos da coleção

- products.size();

8- **keySet()**

Retorna uma lista Set (única) iterando sobre todas as keys da coleção

- products.keySet();

9- **for-each**

E o método mais comum para percorrer coleções de Map é o laço for-each

- products.forEach((key,value) -> System.out.println ("K:" + key + "V:" + value));

Principais aplicações

1-Map<T,S> map = new HashMap<>();

- Desordenado
- Permite apenas chaves únicas
- Alto desempenho para inserção, busca e remoção de elementos

Set

A interface Set é uma coleção capaz de armazenar elementos em lista, mas sem permitir que elementos sejam duplicados (iguais).

Principais Características

- Não permite elementos duplicados
- Não possui limite fixo de elementos

Principais Aplicações

1- Set<T> hashSetList = new HashSet<>();

- Não garante a ordem dos elementos (aleatória)

2- Set<T> linkedSetList = new LinkedHashSet<>();

- Ordenada por ordem de inserção

3- Set<T> treeSetList = new TreeSet<>();

- Ordenada por um critério natural ou por um Comparator

Comparação de classes

Comparator (moderno)

A **interface** Comparator (Comparador), pertence ao pacote **java.util**

Quando implementada, permite definir múltiplos critérios de comparação para elementos de uma classe.

Diferente da interface Comparable, que define a ordem natural diretamente na classe, Comparator é implementada separadamente, proporcionando maior flexibilidade.

Para utilizá-la, podemos sobrescrever o método **compare(T o1, T o2)** na classe que implementa Comparator.

Este método define a lógica de comparação entre dois objetos. Com ele, coleções podem ser ordenadas de forma personalizada, utilizando métodos como Collections.sort(collection, comparator) ou List.sort(comparator).

Outra utilização mais moderna e eficiente é a utilização de expressões lambda.

Exemplo:

- products.sort((p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()));

Comparable (antigo)
A **interface** Comparable, traduzida para Comparável, pertence ao pacote **java.lang**
Quando implementada, permite que elementos de uma classe sejam ordenados e comparados de acordo com um critério natural definido
Dessa forma, somos obrigados a sobrescrever (polimorfismo) o método **compareTo(Type object)** na classe que implementa Comparable.
Este método permite que, a partir da lógica de comparação definida, coleções de objetos sejam ordenadas.
Definido o critério de comparação, o método **Collections.sort(collection_name)** permite a ordenação de coleções
Exemplo de utilização:

- **String**:
  ```
  public class Product implements Comparable<Product> {
  //
  @Override
  public int compareTo(Product otherProduct) {
          return this.getName().compareTo(otherProduct.getName());
  }
  }
  ```
- Tipos primitivos(**Wrapper**)
  ```
  public class Product implements Comparable<Product> {
  //
  @Override
  public int compareTo(Product otherProduct) {
          return Double.compare(this.getPrice(),otherProduct.getPrice());
          // caso a variável seja double e não Double
          return this.getPrice().compareTo(otherProduct.getPrice());
          // caso já seja Double
  ```
- Main
  ```
  //
  List<Product> products = new ArrayList<>();
  Collections.sort(products);
  System.out.println(products);
  ```