
FIAP

Domain Driven Design
using Java

AGENDA

1. Collections
2. Exercícios

Collections

-> Introdução às Coleções

- As coleções em Java são fundamentais para armazenar, manipular e acessar grupos de objetos.
- A principal interface raiz da hierarquia de coleções é *java.util.Collection*.
- As coleções Java são divididas em vários tipos principais, cada uma com suas próprias características e usos.

Principais Interfaces

- ***Collection***: Interface raiz da hierarquia de coleções. Estende a interface *Iterable*.
- ***List***: Uma coleção ordenada que permite elementos duplicados. Exemplos: *ArrayList e LinkedList.*
- ***Set*****:** Coleção que não permite elementos duplicados. Exemplos: *HashSet, LinkedHashSet e TreeSet.*
- ***Queue*****:** Coleção usada para manter elementos antes do processamento. Exemplos: LinkedList e PriorityQueue.
- ***Map*****:** Estrutura que associa chaves a valores. Exemplos: *HashMap, LinkedHashMap, TreeMap e Hashtable.*

Listas (*List*)

- ***ArrayList***: Baseada em um array dinâmico, permite acesso rápido aos elementos.

```
List<String> arrayList = new ArrayList<>();
arrayList.add("Elemento 1");
arrayList.add("Elemento 2");
```

- ***LinkedList***: Baseada em uma lista duplamente ligada, permite inserções/remover rápidas.

```
List<String> linkedList = new LinkedList<>();
linkedList.add("Elemento 1");
linkedList.add("Elemento 2");
```

Conjuntos (*Set*)

- ***HashSet***: Baseado em uma tabela hash, não mantém a ordem dos elementos.

```
Set<String> hashSet = new HashSet<>();
hashSet.add("Elemento 1");
hashSet.add("Elemento 2");
```

- ***LinkedHashSet***: Mantém a ordem de inserção dos elementos.

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Elemento 1");
linkedHashSet.add("Elemento 2");
```

Conjuntos (*Set*)

- ***TreeSet***: Baseado em uma árvore de navegação (Red-Black Tree), mantém os elementos ordenados.

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("Elemento 1");
treeSet.add("Elemento 2");
```

Filas (*Queue*)

- ***LinkedList***: Suporta inserção e remoção de elementos de ambos os extremos.

```
Queue<String> queue = new LinkedList<>();
queue.add("Elemento 1");
queue.add("Elemento 2");
```

- ***PriorityQueue***: Mantém os elementos em uma ordem específica.

```
Queue<String> priorityQueue = new PriorityQueue<>();
priorityQueue.add("Elemento 1");
priorityQueue.add("Elemento 2");
```

Filas (*Queue*)

- ***LinkedList***: Suporta inserção e remoção de elementos de ambos os extremos.

```
Queue<String> queue = new LinkedList<>();
queue.add("Elemento 1");
queue.add("Elemento 2");
```

- ***PriorityQueue***: Mantém os elementos em uma ordem específica.

```
Queue<String> priorityQueue = new PriorityQueue<>();
priorityQueue.add("Elemento 1");
priorityQueue.add("Elemento 2");
```

Mapas (Map)

- ***HashMap***: Baseado em uma tabela hash, permite associações chave-valor.

```
Map<String, String> hashMap = new HashMap<>();
hashMap.put("Chave 1", "Valor 1");
hashMap.put("Chave 2", "Valor 2");
```

- ***LinkedHashMap***: Mantém a ordem de inserção das chaves.

```
Map<String, String> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("Chave 1", "Valor 1");
linkedHashMap.put("Chave 2", "Valor 2");
```

Mapas (Map)

- ***TreeMap***: Mantém as chaves ordenadas.

```
Map<String, String> treeMap = new TreeMap<>();
treeMap.put("Chave 1", "Valor 1");
treeMap.put("Chave 2", "Valor 2");
```

Outras Coleções

- ***Stack***: Representa uma pilha *(LIFO - Last In, First Out*).

```
Stack<String> stack = new Stack<>();
stack.push("Elemento 1");
stack.push("Elemento 2");
```

- ***Vector***: Similar ao *ArrayList*, mas é sincronizado.

```
Vector<String> vector = new Vector<>();
vector.add("Elemento 1");
vector.add("Elemento 2");
```

Algoritmos de Coleções

- **Ordenação**: Utilizando a classe Collections.

```
List<String> list = new ArrayList<>(Arrays.asList("Banana", "Apple", "Or
Collections.sort(list);
```

- **Busca**: Utilizando binarySearch.

```
int index = Collections.binarySearch(list, "Apple");
```

- **Reversão**: Revertendo a ordem dos elementos.

```
Collections.reverse(list);
```

Streams – Programação funcional

- API de Streams (Java 8): Permite processar coleções de maneira funcional e paralela.

```
List<String> list = Arrays.asList("a", "b", "c", "d");
list.stream().filter(s -> s.startsWith("a")).forEach(System.out::printlr
```

Exercícios

1. Cria uma classe pessoa com os atributos nome, documento e idade.
2. Crie uma classe para execução dos Testes.
3. Crie um método que receba uma lista de pessoas, adiciona uma pessoa na lista e retorna a lista preenchida utilizando o List.
4. Crie um método que receba a mesma lista e remova os duplicados utilizando o Set.
5. Crie um método que receba uma pessoa e a adiciona num mapa, colocando como chave o número do documento. Depois, obtenha essa pessoa adicionada ao mapa e imprima os dados no console.
6. Crie um método que ordene as listas em ordem crescente e decrescente, baseado na escolha.


FIAP