Coleções

Coleções

As coleções em Java fazem parte do pacote java.util e oferecem estruturas de dados para armazenar e manipular conjuntos de elementos de forma eficiente.
As principais interfaces são **List**, **Set** e **Map** e cada uma delas tem uma característica específica que nos leva a decidir pela sua utilização

Podemos ter acesso à documentação oficial através deste link.

Vejamos as diferenças e particularidades de cada uma das interfaces

List

A interface **List** representa uma coleção ordenada que permite elementos duplicados. É recomendada quando a ordem de inserção importa.

Suas implementações principais são:

- **ArrayList**: baseada em array dinâmico, promove acesso rápido por índice;
- **LinkedList**: baseada em lista encadeada, eficiente para quando há muitas inserções e remoções de elementos.

```
List<String> funcionarios = new ArrayList<>();
funcionarios.add("João");
funcionarios.add("Maria");
funcionarios.add("João");
```

.add inclui elementos na lista

Set

A interface **Set** é projetada para armazenar elementos únicos. A ordem nem sempre importa, dependendo da implementação.

Suas implementações principais são:

- **HashSet**: baseado em tabelas hash e a ordem de inserção não é garantida
- **LinkedHashSet**: mantém a ordem de inserção
- **TreeSet**: mantém os elementos ordenados.

```
Set<String> produtos = new HashSet<>();
produtos.add("Água");
produtos.add("Coca-cola");
produtos.add("Água"); //Ignorado, pois não aceita duplicados
```

Map

Um **Map** é uma estrutura chave-valor

Suas implementações principais são:

- **HashMap**: baseado em tabelas hash, sem ordem garantida
- **LinkedHashMap**: mantém a ordem de inserção
- **TreeMap**: mantém as chaves ordenadas

```
Map<Integer, String> clientes = new HashMap<>();
clientes.put(1, "Maria");
clientes.put(2, "Marcos");
clientes.put(3, "Ana");
```

alura

Streams

Streams

As partir do Java 8, a API de Streams permite manipular coleções de forma declarativa e eficiente.

As operações em Streams são divididas em intermediárias e terminais:

- **Intermediárias**: Retornam um novo Stream e podem ser encadeadas.
- **Terminais**: Produzem um resultado final e encerram o fluxo.

Streams

Principais operações **intermediárias**:

- filter(Predicate<T>): Filtra elementos com base em uma condição.
- map(Function<T, R>): Transforma os elementos.
- distinct(): Remove elementos duplicados.
- sorted(): Ordena os elementos.
- limit(n): Limita a quantidade de elementos.

Streams

Principais operações **terminais**:

- forEach(Consumer<T>): Executa uma ação para cada elemento.
- collect(Collectors.toList()): Coleta os elementos em uma nova coleção.
- count(): Conta os elementos do Stream.
- findFirst(): Retorna o primeiro elemento encontrado.
- allMatch(Predicate<T>): Verifica se todos os elementos atendem a uma condição.

Streams

Exemplo:

```
List<String> funcionarios = List.of("Ana", "Bruno", "Carlos", "Amanda");
List<String> funcionariosLetraA = funcionarios.stream()
        .filter(f -> f.startsWith("A"))
        .collect(Collectors.toList());
```

.collect pega os elementos filtrados e coleta para a nova coleção *funcionariosLetraA*

.filter filtra os elementos da coleção *funcionarios* cujo nome inicia com a letra A

Streams

Exemplo:

```
double totalComissao = valorVendas.stream()
        .map(v -> v * 0.05)
        .reduce(0.0, Double::sum);
```

.reduce pé uma operação terminal que reduz a Stream a um único valor, somando todos os valores de comissão calculados no map

.map pega cada elemento da coleção *valorVendas* e multiplica por 0.05, para calcular a comissão, gerando um novo Stream<Double>

Compartilhe um resumo de seus novos conhecimentos em suas redes sociais.
#aprendizadoalura

alura | Escola Programação