

# Coleções em Java

Prof. Pedro Pongelupe



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS  
Departamento de Ciência da Computação

# Sumário

- 1 Coleções
  - Java Collections Framework
- 2 Interfaces
  - Collection
  - Set e SortedSet – conjuntos
  - List – listas e Queue – filas
  - Map – mapeamentos
- 3 Streams e pipelines
  - Definição
  - Exemplo Estoque



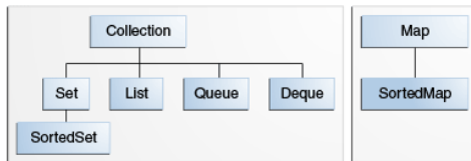
# Collections

## Collections

Uma coleção é um objeto que agrupo múltiplos objetos, como um *container*.

## Java Collections Framework

É uma arquitetura unificada para representação e manipulação de coleções, independentes de implementação.

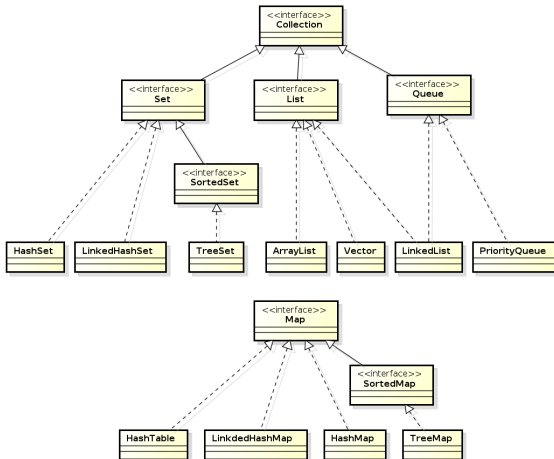


1

<sup>1</sup>Oracle. *Collections: The Java Tutorial*. 2016.



# Java Collections Framework





# Sintaxe

- Exemplo de criação de coleção:

```
List<String> list = new ArrayList<String>();
```

- Processamento:

- for-each

```
for (String s: list ) System.out.println(s);
```

- Iterador

```
Iterator<?> it = list.iterator();  
while( it.hasNext())  
    if (!cond(it.next()))  
        it.remove();
```



# Interface Collection<E>

- Inclusão / remoção:

```
boolean add(E elemento)  
boolean remove(E elemento)
```

- Consultas:

```
int size()  
boolean isEmpty()  
boolean contains(Object elemento)  
Iterator<E> iterator()  
Object[] toArray()
```

- Operações com grupos:

```
boolean containsAll(Collection<?> coleção)  
boolean addAll(Collection<? extends E> coleção)  
void clear()  
void removeAll(Collection<?> coleção)  
void retainAll(Collection<?> coleção)
```



# Interface Set<E> – conjuntos de elementos

- Conjuntos: não aceitam elementos duplicados.
- Principais métodos:
  - Herdados da interface Collection.
- Implementações:
  - Classe HashSet – acesso mais rápido.
  - Classe TreeSet – acesso ordenado.
  - Classe LinkedHashSet – mais versátil.
- Considerações:
  - Dois conjuntos são iguais se contiverem os mesmos elementos (determinado através dos métodos equals() e hashCode()).



# Interface SortedSet<E> – conjunto ordenado

- Principais métodos:

E first()

E last()

SortedSet<E> headSet(E aoElemento)

SortedSet<E> subSet(E doElemento, E aoElemento)

SortedSet<E> tailSet(E doElemento)

Comparator<? **super** E> comparator()





# Exemplo de conjuntos

```
public class ExemploSet {  
    public static void main( String[] args ) {  
  
        Set<String> conjunto = new HashSet<>();  
  
        conjunto.add( "Bernardo" );  
        conjunto.add( "Carolina" );  
        conjunto.add( "Felipe" );  
        conjunto.add( "Carolina" );  
        conjunto.add( "Ana" );  
  
        System.out.println( conjunto );  
  
        Set<String> conjuntoOrdenado =  
            new TreeSet<>( conjunto );  
  
        System.out.println( conjuntoOrdenado );  
    }  
}
```



## Solução parcial: aluno

```
public class Aluno {  
    private final String matricula;  
    private String nome;  
    private int idade;  
  
    public Aluno(String matricula, String nome, int idade) {  
        this.matricula = matricula;  
        this.nome = nome;  
        this.idade = idade;  
    }  
  
    /** getters e setters */  
  
}
```



## Solução parcial: Aluno

```
public static void main(String[] args) {  
    var aluno1 = new Aluno("123", "Pedro", 25);  
    var aluno2 = new Aluno("123", "Pedro", 25);  
    System.out.println(aluno1.equals(aluno2));  
  
    Set<Aluno> set = new HashSet<>();  
    set.add(aluno1);  
    set.add(aluno2);  
  
    System.out.println(set.size());  
}
```



## Interface List<E> – listas

- Coleção indexada com possibilidade de chaves duplicadas
- Principais métodos:

```
void add( int índice , E elemento )  
boolean add( E elemento )  
boolean addAll( int índice , Collection<? extends E> coleção )  
  
E get( int índice )  
E set( int índice , E element )  
int indexOf( Object elemento )  
int lastIndexOf( Object elemento )  
  
E remove( int índice )  
List<E> subList( int índiceInicial , int índiceFinal )
```



# Interface ListIterator<E> – iterador de listas

```
ListIterator <E> listIterator ()
```

```
ListIterator <E> listIterator (int índiceInicial )
```

- Principais métodos:

```
void add( E elemento )
```

```
void set( E elemento )
```

```
void remove()
```

```
boolean hasPrevious()
```

```
boolean hasNext()
```

```
E previous()
```

```
E next()
```

```
int nextIndex()
```

```
int previousIndex()
```



# Implementações de listas

- Classe `ArrayList<E>` – semelhante a vetores dinâmicos.
  - Implementa os métodos da interface.
- Classe `LinkedList<E>` – manipulação sequencial de elementos (filas, pilhas, dequeues, etc.).
  - Implementa métodos adicionais, além dos da interface:

```
void addFirst( E elemento )  
void addLast( E elemento )  
E getFirst()  
E getLast()  
Object removeFirst()  
Object removeLast()
```



## Exemplo de listas

```
public class ExemploDeListas {  
    public static void main(String[] args) {  
        List<String> lista = new ArrayList<String>();  
        lista.add("Bernardo");        lista.add("Carolina");  
        lista.add("Felipe");          lista.add("Carolina");  
        lista.add("Clara");  
        System.out.println(lista);  
        System.out.println("1: " + lista.get(2));  
        LinkedList<String> fila = new LinkedList<String>();  
        fila.addFirst("Bernardo");    fila.addFirst("Carolina");  
        fila.addFirst("Felipe");      fila.addFirst("Ana");  
        fila.addFirst("Clara");  
        System.out.println(fila);  
        fila.removeLast();  
        fila.removeLast();  
        System.out.println(fila);  
    }  
}
```



## Interface Queue<E> – filas

- Coleção baseada em filas, com possibilidade de prioridades e bloqueios
- Principais métodos:

E element() // retorna 1o elem. da fila , sem removê-lo.  
**boolean** offer(E elemento) // insere se possível.  
**boolean** add(E elemento) // insere ou lança exceção.  
E peek() // recupera elem., mas não remove ou retorna null.  
E poll() // recupera e remove ou retorna null.  
E remove() // recupera e remove ou lança exceção.

- Implementações de filas:
  - Classe PriorityQueue<E> – semelhante às listas, só que implementadas em Heap.
    - Elem. ordenados por ordenação natural ou por Comparator .
    - Objetos devem ser comparáveis (**interface** Comparable<E>):  
**public int** compareTo( E e ) **throws** ClassCastException





## Interface Map<K,V> – mapeamentos

- Associações de chaves (*K – Keys*) e valores (*V – Values*).
- Principais métodos para alteração:
  - `V put( K chave, V valor )`
  - `V remove( K chave )`
  - `void putAll( Map<? extends K, ? extends V> mapeamento )`
  - `void clear()`
- Principais métodos para consulta:
  - `V get( K chave )`
  - `boolean containsKey( Object chave )`
  - `boolean containsValue( Object valor )`
  - `int size()`
  - `boolean isEmpty()`
- Principais métodos para grupos:
  - `Set<K> keySet()`
  - `Collection<V> values()`
  - `Set<Map.Entry<K,V>> entrySet()`



# Map.Entry<K,V> – elementos de mapeamentos

- Representa um par chave-valor.
- Principais métodos:

```
boolean equals( Object objeto )  
K getKey ();  
V getValue ();  
V setValue( V valor );
```

- Implementações de mapeamentos:
  - **class** HashMap<K,V> – agilidade, permite `nulls`.
  - **class** TreeMap<K,V> – ordenação (árvore balanceada).
  - **class** LinkedHashMap<K,V> – ordem de iteração previsível.



# SortedMap<K,V> – mapeamentos ordenados

- Implementado pelo TreeMap<K, V>.

- Principais métodos:

```
Comparator<? super K> comparator();  
SortedMap<K,V> headMap( K ateChave );  
SortedMap<K,V> subMap( K daChave, K ateChave );  
SortedMap<K,V> tailMap( K daChave );  
K firstKey();  
K lastKey();
```



## Exemplo de mapeamentos

```
public static void main(String[] args) {  
    Map<String, Integer> mapH = new HashMap<>();  
    for (int i = 0; i < args.length; i++) {  
        String chave = args[i];  
        Integer frequencia = mapH.get(chave);  
        if (frequencia == null) {  
            frequencia = 1;  
        } else {  
            frequencia++;  
        }  
  
        mapH.put(chave, frequencia);  
    }  
    System.out.println(mapH);  
  
    var mapOrd = new TreeMap<>(mapH);  
    System.out.println(mapOrd);  
}
```



# Algoritmos de Coleções

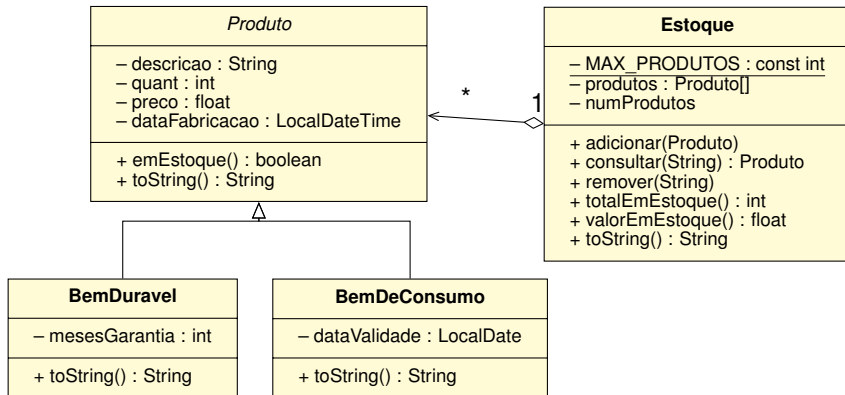
- Principais algoritmos aplicados a coleções (classe Collections):

```
sort() // ordena uma lista
shuffle() // embaralha uma lista
reverse() // inverte a ordem dos elementos em uma lista
fill() // preenche uma coleção com determinado elemento
copy() // copia os elementos de uma coleção para outra
swap() // inverte dois elementos em uma lista
binarySearch() // busca binária em uma lista ordenada
frequency() // calcula o nº de vezes que elem. aparece numa coleção
disjoint() // verifica se duas listas não têm elementos em comum
min() // acha o menor valor em uma coleção
max() // acha o maior valor em uma coleção
```



# Java *pipelines* e *streams*

- Para aprendermos *streams*, precisamos relembrar a hierarquia de produtos.





## Java *pipelines* e *streams*

- **Stream** é uma sequência de elementos. Ao contrário da *Collection*, não é uma estrutura de dados que armazena elementos, mas transporta valores através de um *pipeline*, como um fluxo de dados.
- **Pipeline** é uma sequência de operações sobre *streams*.
- O exemplo imprime os produtos com *preço* > 100.0.

```
List<Produto> produtos = new ArrayList<Produto>();  
produtos.stream()  
    .filter(prod -> prod.getPreco() > 100)  
    .forEach(prod -> System.out.println(prod));
```



# Java *pipelines* e *streams*

- Compare o exemplo anterior com:

```
for (Produto prod: produtos) {  
    if (prod.getPreco() > 100) {  
        System.out.println(prod);  
    }  
}
```

- Perceba que o processamento em *pipeline* executou operações em sequência sobre um fluxo de dados:
  - 1 Filtrou a lista mantendo apenas os produtos com preço maior que 100.0.
  - 2 Imprimiu cada produto restante após a filtragem.





# Java *pipelines* e *streams*

- O exemplo a seguir calcula a quantidade média por BemDeConsumo no estoque, pelo *pipeline*:
  - 1 converte lista de produtos em um *stream*.
  - 2 filtra o *stream* mantendo os produtos do tipo BemDeConsumo.
  - 3 mapeia o resultado para `int`, baseado no atributo `quant` (retorna `IntStream`).
  - 4 calcula a média da lista de inteiros (retorna `OptionalDouble`).
  - 5 retorna a média como `double`.

```
double media = produtos
    .stream()
    .filter(p -> p instanceof BemDeConsumo)
    .mapToInt(Produto::getQuant)
    .average()
    .getAsDouble();
```



## Exemplo: Estoque com *collections* e *streams*

### Java array

```
public class Estoque {  
    private static final int MAX_PRODUTOS = 100;  
    private Produto[] listaDeProdutos;  
    private int numProdutos;  
}
```

### Collections e Streams

```
public class Estoque {  
    List<Produto> listaDeProdutos;  
  
    public List<Produto> getListaDeProdutos() {  
        return listaDeProdutos;  
    }  
}
```



## Exemplo: Estoque com *collections* e *streams*

### Java array

```
public void adicionar(Produto p) {  
    if (numProdutos < MAX_PRODUTOS) {  
        listaDeProdutos[numProdutos++] = p;  
    }  
}
```

### Collections e Streams

```
public void adicionar(Produto p) {  
    listaDeProdutos.add(p);  
}
```



# Exemplo: Estoque com *collections* e *streams*

## Java array

```
public void remover(String descricao) {  
    for (int pos = 0; pos < numProdutos; pos++) {  
        if (descricao.equals(listaDeProdutos[pos].getDescricao())) {  
            for (int i = pos + 1; i < numProdutos; i++)  
                listaDeProdutos[i - 1] = listaDeProdutos[i];  
            listaDeProdutos[numProdutos - 1] = null;  
            numProdutos--;  
        }  
    }  
}
```

## Collections e Streams

```
public void remover(String descricao) {  
    listaDeProdutos.removeIf((prod) -> descricao.equals(prod.getDescricao()));  
}
```



# Exemplo: Estoque com *collections* e *streams*

## Java array

```
public Produto consultar(String descricao) {  
    for (int pos = 0; pos < numProdutos; pos++) {  
        if (descricao.equalsIgnoreCase(listaDeProdutos[pos].getDescricao())) {  
            return listaDeProdutos[pos];  
        }  
    }  
    return null;  
}
```

## Collections e Streams

```
public Produto consultar(String descricao) {  
    for (Produto p : listaDeProdutos) {  
        if (descricao.equalsIgnoreCase(p.getDescricao()))  
            return p;  
    }  
    return null;  
}
```



## Exemplo: Estoque com *collections* e *streams*

### Java array

```
public int totalEmEstoque() {  
    int total = 0;  
    for (int i = 0; i < numProdutos; i++)  
        total += listaDeProdutos[i].getQuant();  
    return total;  
}
```

### Collections e Streams

```
public int totalEmEstoque() {  
    return listaDeProdutos.stream()  
        .mapToInt(Produto::getQuant)  
        .reduce(0, (x, y) -> x + y);  
}
```



## Exemplo: Estoque com *collections* e *streams*

### Java array

```
public float valorEmEstoque() {  
    float valor = 0;  
    for (int i = 0; i < numProdutos; i++)  
        valor += listaDeProdutos[i].getQuant() *  
                listaDeProdutos[i].getPreco();  
    return valor;  
}
```

### Collections e Streams

```
public float valorEmEstoque() {  
    return listaDeProdutos.stream()  
        .map((pd) -> pd.getQuant() * pd.getPreco())  
        .reduce(0.0F, (x, y) -> x + y);  
}
```



## Exemplo: Estoque com *collections* e *streams*

### Java array

```
public String toString() {  
    StringBuilder valor = new StringBuilder();  
    for (int i = 0; i < numProdutos; i++) {  
        valor.append(listaDeProdutos[i] + "\n");  
    }  
    return valor.toString();  
}
```

### Collections e Streams

```
public String toString() {  
    return listaDeProdutos.stream()  
        .map((prod) -> prod.toString() + "\n")  
        .reduce("", (x, y) -> x + y);  
}
```





## Exemplo: Estoque com *collections* e *streams*

- O método a seguir ordena o estoque em ordem alfabética de descrição.
- Ele não utiliza *streams*.
- Ele implementa a interface `Comparator` em uma classe anônima passada diretamente como parâmetro.

```
public void ordenar() {  
    listaDeProdutos.sort(new Comparator<Produto>() {  
        @Override  
        public int compare(Produto o1, Produto o2) {  
            return (o1.getDescricao().compareTo(o2.getDescricao()));  
        }  
    });  
}
```



## Exemplo: Estoque com *collections* e *streams*

- O método método ordenar poderia ser implementado na forma de *streams*.

```
public void ordenar() {  
    listaDeProdutos =  
        listaDeProdutos.stream()  
            .sorted(  
                (x, y) -> x.getDescricao().compareTo(y.getDescricao())  
            )  
            .collect(toList());  
}
```



## Exemplo: Estoque com *collections* e *streams*

- É possível se criar um método parametrizável.
- Esse método recebe dois parâmetros: uma condição (`Predicate`) e um método de comparação (`Comparator`).
- Retorna a lista de produtos filtrada pela condição e ordenada pelo critério definido.

```
public List<Produto> ordenarStream(Predicate<Produto> p,  
                                   Comparator<Produto> c) {  
    return listaDeProdutos.stream()  
        .filter(p)  
        .sorted(c)  
        .collect(toList());  
}
```



## Exemplo: Estoque com *collections* e *streams*

- Para usar os novos métodos criados na classe Estoque.

```
public static void main(String args[]) {  
    ...  
    estoque.ordenar();  
  
    List<Produto> filtrado = estoque.ordenarStream(  
        ((pf) -> pf.getQuant() > 100),  
        ((p1, p2) -> Float.compare(p1.getPreco(), p2.getPreco())));  
  
    filtrado.forEach(prod -> System.out.println(prod));  
  
    double media = estoque.getListaDeProdutos()  
        .stream()  
        .filter(p -> p instanceof BemDeConsumo)  
        .mapToInt(Produto::getQuant)  
        .average()  
        .getAsDouble();  
  
    System.out.println("Quantidade média por produto: " + media);  
    System.out.println("Total em estoque: " + estoque.totalEmEstoque());  
    System.out.println("Valor em estoque: " + estoque.valorEmEstoque());  
}
```



# Obrigado!!

Muito obrigado pela atenção! Alguma dúvida? Bora praticar!!!

*"Uma frase muda o fim do filme."*