Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Ciência da Computação
Artigo Científico

Java Collections

Autor: Eduardo Salomão, Gabriel Ribeiro
Orientador: João Pedro Macleure

Brasília - DF
2023

Eduardo Salomão, Gabriel Ribeiro

Java Collections

Artigo apresentado ao curso de graduação em Ciência do computação da Universidade Católica de Brasília, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Orientador: Prof. João Pedro Macleure

Brasília
2023

RESUMO

No contexto da programação em Java, diversas interfaces e classes desempenham papéis fundamentais no manejo de coleções e estruturas de dados. A interface Iterable é essencial para permitir iteração eficiente de elementos em coleções, enquanto a interface Collection define um conjunto de métodos padrão para operações com coleções de objetos, como adição, remoção e iteração. A interface Set representa uma coleção de elementos únicos, sem permitir duplicações, e a interface List representa uma coleção ordenada que permite elementos duplicados e acesso por índices. A interface Queue é usada para modelar filas de elementos, seguindo o princípio FIFO, e a interface Map representa uma coleção de pares chave-valor, com chaves exclusivas.

Além das interfaces, várias classes desempenham funções cruciais. As classes AbstractSet e AbstractList são bases abstratas para a implementação de conjuntos e listas que implementam as interfaces Set e List. HashSet é uma implementação de conjunto que utiliza uma tabela de dispersão para armazenar elementos eficientemente, enquanto LinkedList armazena elementos em uma lista duplamente encadeada, proporcionando boa eficiência para inserções e remoções. A classe Vector é uma implementação de lista thread-safe, embora menos eficiente que ArrayList. ArrayList armazena elementos em um array dinâmico e é ideal para acesso rápido por índices. A classe Dictionary, embora menos comum, representa uma tabela de pares chave-valor. As classes AbstractMap e HashTable são bases para a implementação de mapas que implementam a interface Map. Por fim, HashMap é uma implementação altamente eficiente da interface Map, que armazena pares chave-valor em uma tabela de dispersão, sendo amplamente usada em diversos cenários de programação em Java.

Palavra-chave: Palavras-chave: Iterable; Collection; Map; Classe; Interface; Java.

Abstract

In the context of Java programming, various interfaces and classes play fundamental roles in handling collections and data structures. The Iterable interface is essential for enabling efficient iteration of elements in collections, while the Collection interface defines a set of standard methods for operations on collections of objects, such as addition, removal, and iteration. The Set interface represents a collection of unique elements, preventing duplicates, and the List interface represents an ordered collection that allows duplicate elements and supports indexed access. The Queue interface is used to model queues of elements, following the FIFO (First-In-First-Out) principle, and the Map interface represents a collection of key-value pairs with unique keys.

In addition to the interfaces, several classes perform crucial functions. The AbstractSet and AbstractList classes serve as abstract bases for implementing sets and lists that adhere to the Set and List interfaces, respectively. HashSet is a set implementation that efficiently stores elements using a hash table. LinkedList stores elements in a doubly-linked list, providing good efficiency for insertions and removals. The Vector class is a thread-safe list implementation, albeit less efficient than ArrayList. ArrayList stores elements in a dynamic array and is ideal for fast indexed access. The Dictionary class, although less common, represents a table of key-value pairs. The AbstractMap and HashTable classes serve as bases for implementing maps that conform to the Map interface. Finally, HashMap is a highly efficient implementation of the Map interface, storing key-value pairs in a hash table and widely used in various Java programming scenarios.

Keyword: Keywords: Iterable; Collection; Map; Classe; Interface; Java.

Sumário

1. INTRODUÇÃO

1.1. JAVA COLLECTIONS

Java Collections é um framework que fornece classes e interfaces para lidar com coleções de objetos, como listas, conjuntos e mapas. As coleções são úteis para armazenar, manipular e recuperar dados de forma eficiente. Os Mapas, em particular, são estruturas de dados que associam chaves a valores e são amplamente usados para mapear valores únicos para chaves.

2. INTERFACES

2.1. ITERABLE<T>

A interface Iterable em Java é fundamental para permitir a iteração eficiente de elementos em coleções. Essa interface define um único método, iterator(), que retorna um objeto do tipo Iterator. O Iterator é usado para percorrer sequencialmente os elementos de uma coleção, permitindo que você acesse e processe cada elemento individualmente.
Ao implementar a interface Iterable, uma classe permite que seus objetos sejam usados em estruturas de controle de iteração, como o loop for-each Isso simplifica a iteração em coleções, tornando o código mais legível e eficiente.
Em resumo, a interface Iterable não representa uma coleção por si só, mas é essencial para habilitar a iteração sobre elementos em coleções. Ela desempenha um papel fundamental no modelo de iteração do Java, tornando mais fácil trabalhar com objetos que armazenam conjuntos de dados.

2.1.1. Subinterfaces do Iterable

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Collection<E>, Deque<E>, DirectoryStream<T>, List<E>, NavigableSet<E>, Path, Queue<E>, SecureDirectoryStream<T>, Set<E>, SortedSet<E>, TransferQueue<E>

2.1.2. Classes de implementação

AbstractCollection,AbstractList,AbstractQueue,AbstractSequentialList,AbstractSet,ArrayBlockingQueue,ArrayDeque,ArrayList, AttributeList, BatchUpdateException, BeanContextServicesSupport,BeanContextSupport, oncurrentHashMap.KeySetView, ConcurrentLinkedDeque,ConcurrentLinkedQueue,ConcurrentSkipListSet,CopyOnWriteArrayList,CopyOnWriteArraySet,DataTruncation, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList,RowSetWarning,SerialException,ServiceLoader,SQLClientInfoException, SQLDataException, SQLException, SQLFeatureNotSupportedException, SQLIntegrityConstraintViolationException,SQLInvalidAuthorizationSpecException,SQLNonTransientConnectionException,SQLNonTransientException,SQLRecoverable

Exception,SQLSyntaxErrorException,SQLTimeoutException,SQLTransactionRollbackException, SQLTransientConnectionException, SQLTransientException, SQLWarning, Stack, SyncFactoryException, SynchronousQueue, SyncProviderException, TreeSet, Vector

2.1.3. Métodos do Iterable

- Iterator: Iterator<T> iterator(), retorna um iterável de elementos do tipo T.

- forEach: default void forEach(Consumer<? super T> action), executa a ação fornecida para cada elemento do Iterable até que todos os elementos tenham sido processados ou a ação gere uma exceção. A menos que especificado de outra forma pela classe de implementação, as ações são executadas na ordem de iteração (se uma ordem de iteração for especificada).

- spliterator: default Spliterator<T> spliterator(), cria um Spliterator sobre os elementos descritos por este Iteravel.

2.2. COLLECTION<E>

A interface Collection em Java é uma parte fundamental do framework Collections. Ela define um conjunto de métodos padrão que todas as classes que representam coleções devem implementar. A Collection fornece operações para adicionar, remover, verificar a presença e percorrer elementos em uma coleção de objetos.
Essa interface serve como a raiz da hierarquia de coleções, e suas principais subinterfaces incluem List, Set, e Queue. Cada uma dessas subinterfaces tem características específicas. Por exemplo, List permite armazenar elementos em uma ordem específica e permite elementos duplicados, enquanto Set não permite elementos duplicados. Por outro lado, Queue é usada para representar uma fila de elementos, onde o primeiro elemento a ser inserido é o primeiro a ser removido.
A interface Collection simplifica o gerenciamento de dados em Java, fornecendo um conjunto comum de métodos que podem ser usados em diferentes tipos de coleções. Isso torna mais fácil desenvolver código genérico e reutilizável, pois você pode tratar diferentes tipos de coleções da mesma maneira, desde que implementem a interface Collection.

2.2.1. Superinterfaces

Iterable<E>

2.2.2. Subinterfaces

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

2.2.3. Classes de implementação

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue,ArrayDeque,ArrayList,AttributeList,BeanContextServicesSupport,BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue,ConcurrentSkipListSet,CopyOnWriteArrayList,CopyOnWriteArraySet,DelayQueue,EnumSet,HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue,LinkedHashSet,LinkedList,LinkedTransferQueue,PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

2.2.4. Métodos

Além do método forEach que é herdado da interface Iterable, a interface Collection possui os seguintes métodos:

- Add: boolean add(E e), adiciona a esta coleção o elemento especificado.
- addAll: boolean addAll(Collection<? extends E> c), adiciona todos os elementos da coleção especificada a esta coleção.
- clear: void clear(), remove todos os elementos da coleção.
- contains: boolean contains (Object o), retorna true se a coleção possui o elemento especificado.
- containsAll: boolean containsAll(Collection<?> c), retorna true se esta coleção possui todos os elementos da coleção especificada.
- equals: boolean equals(Object o), compara se o objeto especificado é igual a coleção.
- hashCode: int hashCode(), retorna o valor do código hash para esta coleção.
- isEmpity: boolean isEmpity(), retorna true se a coleção não possui elementos.
- parallelStream: default Stream<E> parallelStream(), retorna um Stream possivelmente paralelo com esta coleção como origem.
- remove: boolean remove(Object o), remove um elemento especificado da coleção, se ele existir.
- removeAll: boolean removeAll(Collection<?> c), remove todos os elementos desta coleção que também estão presentes na coleção especificada.
- removeIf: default boolean removeIf(Predicate<? super E> filter), Remove todos os elementos desta coleção que satisfazem o predicado fornecido.

- retainAll: boolean retainAll(Collection<?> c), retém apenas os elementos desta coleção que estão contidos na coleção especificada.

- Iterator: Iterator<E> iterator(), retorna um iterável para todos os elementos da coleção

- size: int size(), retorna o número de elementos da coleção.

- spliterator: default Spliterator<E> spliterator(), cria um Spliterator sobre os elementos desta coleção.

- stream: default Stream<E> stream(), retorna um Stream sequencial com esta coleção como origem.

- toArray: Object[] toArray(), retorna um array contendo todos os elementos desta coleção.

- toArray: <T> T[] toArray(T[] a), retorna um array contendo todos os elementos desta coleção; o tipo de do array retornado em tempo de execução é o do array especificado.

2.3. SET<E>

A interface Set em Java é uma das principais subinterfaces da interface Collection e define um contrato para coleções que representam conjuntos de elementos únicos, ou seja, que não permitem elementos duplicados. Características-chave da interface Set incluem:

- Unicidade de Elementos: Um Set não permite a inserção de elementos duplicados. Cada elemento é único na coleção.

- Sem Ordem Específica: Ao contrário da interface List, a interface Set não garante uma ordem específica dos elementos. Os elementos não são armazenados em uma sequência específica, embora algumas implementações, como LinkedHashSet, possam manter uma ordem de inserção.

- Operações Básicas: A interface Set herda métodos da interface Collection para adicionar, remover e verificar a presença de elementos na coleção.

- Implementações Comuns: Algumas das implementações mais comuns da interface Set incluem HashSet, TreeSet, LinkedHashSet e EnumSet, cada uma adequada para diferentes cenários de uso.

A interface Set é amplamente usada quando você precisa garantir que os elementos em uma coleção sejam únicos, independentemente da ordem em que foram inseridos. Ela é uma escolha eficaz para verificar a unicidade de elementos e realizar operações conjuntivas em conjuntos de dados.

2.3.1. Superinterfaces

Collection<E>, Iterable<E>

2.3.2. Subinterfaces

NavigableSet<E>, SortedSet<E>

2.3.3. Classes de implementação

AbstractSet,ConcurrentHashMap.KeySetView,ConcurrentSkipListSet,CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

2.3.4. Métodos

Além dos Métodos parallelStream, removeIf, stream herdado da interface Collection e forEach herdado da Iterable, a interface Set possui os seguintes métodos:

- add: boolean add(E e), adiciona o elemento especificado a este conjunto se ainda não estiver presente.
- addAll: boolean addAll (Collection<? extends E> c), adiciona todos os elementos da coleção especificada a este conjunto se eles ainda não estiverem presentes.
- clear: void clear(), remove todos os elementos deste conjunto.
- contains: boolean contains(Object o), retornará true se este conjunto contiver o elemento especificado.
- containsAll: boolean containsAll(Collection<?> c), retornará true se este conjunto contiver todos os elementos da coleção especificada.
- equals: boolean equals(), compara o objeto especificado com este conjunto quanto à igualdade.
- hashCode: int hashCode(), retorna o valor do código hash para este conjunto.
- isEmpity: boolean isEmpity(), retorna true se o conjunto não contém elementos.
- iterator: Iterator<E> iterator(), retorna um iteravel sobre os elementos deste conjunto.

- remove: boolean remove(Object o), remove o elemento especificado deste conjunto.

- removeAll: boolean removeAll(Collection<?> c), remove deste conjunto todos os seus elementos contidos na coleção especificada.

- retainAll: boolean retainAll(Collection<?> c), retém apenas os elementos deste conjunto que estão contidos na coleção especificada.

- size: int size(), retorna o número de elementos desse conjunto.

- spliterator: default Spliterator<E> spliterator(), cria um Spliterator sobre os elementos deste conjunto.

- toArray: Object[] toArray(), retorna um array contendo todos os elementos deste conjunto.

- toArray: <T> T[] toArray(T[] a), retorna um array contendo todos os elementos deste conjunto; o tipo do array retornado em tempo de execução é o do array especificado.

2.4. LIST<E>

A interface List em Java é uma subinterface da interface Collection e define uma coleção ordenada de elementos, onde os elementos podem ser acessados por índices. Características-chave da interface List incluem:

- Ordenação: A ordem dos elementos em uma lista é significativa e é mantida de acordo com a sequência de inserção. Isso permite acessar elementos por índice.

- Duplicação: Listas podem conter elementos duplicados, ou seja, o mesmo elemento pode aparecer mais de uma vez na lista.

- Operações Específicas: Além das operações herdados da interface Collection, a interface List fornece métodos adicionais para acessar, inserir e remover elementos por índice.

- Implementações Comuns: Algumas implementações populares da interface List incluem ArrayList, LinkedList, e Vector.

A interface List é amplamente utilizada quando a ordem dos elementos é importante, ou quando você precisa acessar elementos com base em índices. Ela fornece flexibilidade para armazenar e manipular elementos de forma sequencial, tornando-se uma escolha comum em muitas aplicações Java.

2.4.1. Superinterfaces

Collection<E>, Iterable<E>

2.4.2. Classes de implementação

AbstractList,AbstractSequentialList,ArrayList,AttributeList,CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

2.4.3. Métodos

Além dos métodos parallelStream, removeIf, stream herdados de Collection e forEach herdado de Iterable, a interface List possui os seguintes métodos:

- add: boolean add(E e), a crescenta o elemento especificado ao final desta lista.
- add: void add(int index, E element), insere o elemento especificado na posição especificada nesta lista.
- addAll: boolean addAll(Collection<? extends E> c), acrescenta todos os elementos da coleção especificada ao final desta lista, na ordem em que são retornados pelo iterador da coleção especificada.
- addAll: boolean addAll(int index, Collection<? extends E> c), insere todos os elementos da coleção especificada nesta lista na posição especificada.
- clear: void clear(), remove todos os elementos desta lista.
- contains: boolean contains(Object o), retornará true se esta lista contiver o elemento especificado.
- containsAll: boolean containsAll(Collection<?> c), retornará true se esta lista contiver todos os elementos da coleção especificada.
- equals: boolean equals(Object o), compara o objeto especificado com esta lista quanto à igualdade.
- get: E get(int index), retorna o elemento na posição especificada nesta lista.
- hashCode: int hashCode(), retorna o valor do código hash para esta lista.
- indexOf: int indexOf(Object o), retorna o índice da primeira ocorrência do elemento especificado nesta lista ou -1 se esta lista não contiver o elemento.
- isEmpity: boolean isEmpity(), retorna true se esta lista não contém elementos.
- Iterator: Iterator<E> iterator(), retorna um iterador sobre os elementos desta lista na sequência adequada.
- lastIndexOf: int lastIndexOf(Object o), retorna o índice da última ocorrência do elemento especificado nesta lista ou -1 se esta lista não contiver o elemento.

- listIterator: ListIterator<E> listIterator(), retorna um iterador de lista sobre os elementos desta lista (na sequência adequada).

- listIterator: ListIterator<E> listIterator(int index), retorna um iterador de lista sobre os elementos desta lista (na sequência adequada), começando na posição especificada na lista.

- remove: E remove(int index), remove o elemento na posição especificada nesta lista.

- remove: boolean remove(Object o), remove a primeira ocorrência do elemento especificado desta lista, se estiver presente.

- removeAll: boolean removeAll(Collection<?> c), remove desta lista todos os seus elementos contidos na coleção especificada.

- replaceAll: default void replaceAll(UnaryOperator<E> operator), substitui cada elemento desta lista pelo resultado da aplicação do operador a esse elemento.

- retainAll: boolean retainAll(Collection<?> c), retém apenas os elementos desta lista que estão contidos na coleção especificada.

- set: E set(int index, E element), substitui o elemento na posição especificada nesta lista pelo elemento especificado.

- size: int size(), retorna o número de elementos desta lista.

- sort: default void sort(Comparator<? super E> c), organiza esta lista de acordo com a ordem induzida pelo Comparador especificado.

- spliterator: default Spliterator<E> spliterator(), cria um Spliterator sobre os elementos desta lista.

- subList: List<E> subList(int fromIndex, int toIndex), Retorna uma visualização da parte desta lista entre o especificado fromIndex, inclusive, e toIndex, exclusivo.

- toArray: Object[] toArray(), retorna um array contendo todos os elementos desta lista na sequência adequada (do primeiro ao último elemento).

- toArray: <T> T[] toArray(T[] a), Retorna um array contendo todos os elementos desta lista na sequência correta (do primeiro ao último elemento); o tipo do array retornado em tempo de execução é o do array especificado.

2.5. QUEUE<E>

A interface Queue em Java é uma subinterface da interface Collection que representa uma coleção de elementos ordenados de acordo com o princípio "FIFO" (First-In-First-Out). Isso significa que o primeiro elemento a ser inserido na fila é o primeiro a ser removido. As operações comuns em uma fila incluem enfileirar (adicionar elementos) e desenfileirar (remover elementos). Além disso, a interface Queue oferece métodos como add, offer, remove e poll para essas operações. Implementações comuns incluem LinkedList e PriorityQueue. A interface Queue é frequentemente usada em cenários onde a ordem de processamento é crítica, como em filas de tarefas, processamento de pedidos em sistemas de comércio eletrônico e outros casos em que o processamento de elementos deve seguir uma ordem específica.

2.5.1. Superinterfaces

Collection<E>, Iterable<E>

2.5.2. Subinterfaces

BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

2.5.3. Classes de implementação

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue,DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList,LinkedTransferQueue,PriorityBlockingQueue,PriorityQueue,SynchronousQueue

2.5.4. Métodos

Além dos métodos addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, remove, removeAll, removeIf, retainAll, size, spliterator, stream, toArray, toArray que herdam de Collection e forEach que herda de Iterable, a interface Queue possui os seguintes métodos:

- add: boolean add(E e), insere o elemento especificado nesta fila se for possível fazê-lo imediatamente sem violar as restrições de capacidade, retornando verdadeiro em caso de sucesso e lançando uma IllegalStateException se não houver espaço disponível no momento.
- element: E element(), recupera, mas não remove, o início desta fila.
- offer: boolean offer(E e), insere o elemento especificado nesta fila se for possível fazê-lo imediatamente sem violar as restrições de capacidade.
- peek: E peek(), recupera, mas não remove, o cabeçalho desta fila ou retorna nulo se esta fila estiver vazia.

- pool: E poll(), recupera e remove o cabeçalho desta fila ou retorna nulo se esta fila estiver vazia.

- remove: E remove(), recupera e remove o início desta fila.

2.6. MAP<K, V>

A interface Map em Java é uma parte fundamental do framework Collections e representa uma coleção de pares chave-valor, onde cada chave é única. Principais características da interface Map incluem:

- Associação Chave-Valor: Os elementos em um Map são organizados em pares chave-valor, onde cada chave é usada para recuperar o valor associado.

- Chaves Únicas: Cada chave em um Map é única e não pode ser duplicada.

- Operações de Inserção, Remoção e Recuperação: A interface Map fornece métodos para inserir, remover e recuperar valores com base em suas chaves.

- Implementações Comuns: Algumas das implementações populares da interface Map incluem HashMap, TreeMap, LinkedHashMap e Hashtable.

- Mapeamento Eficiente: A estrutura de dados subjacente em um Map é projetada para permitir uma recuperação eficiente dos valores com base em suas chaves, tornando-a adequada para tarefas como indexação e pesquisa.

A interface Map é amplamente usada para armazenar e recuperar informações associadas a chaves exclusivas, tornando-a útil em uma variedade de cenários, como gerenciamento de configurações, cache, armazenamento de dados em memória e muito mais. É uma estrutura fundamental para manipulação de dados em Java.

2.6.1. Subinterfaces

Bindings,ConcurrentMap<K,V>,ConcurrentNavigableMap<K,V>,LogicalMessageContext,MessageContext,NavigableMap<K,V>,SOAPMessageContext,SortedMap<K,V>

2.6.2. Classes de implementação

AbstractMap,Attributes,AuthProvider,ConcurrentHashMap,ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

2.6.3. Classe/Interface aninhada

static interface Map.Entry<K,V>:

A interface Map.Entry em Java é uma interface aninhada que faz parte da interface Map. Ela é usada para representar um par chave-valor dentro de um mapa. Cada entrada em um mapa é composta por uma chave única e o valor associado a essa chave. A interface Map.Entry define dois métodos principais:

- getKey(): Este método retorna a chave associada à entrada.
- getValue(): Este método retorna o valor associado à chave na entrada.

Por meio desses métodos, é possível acessar tanto a chave quanto o valor de uma entrada no mapa. A interface Map.Entry é frequentemente usada em conjunto com o método entrySet() da interface Map para iterar sobre as entradas de um mapa.

2.6.4. Métodos

- clear: void clear(), remove todos os mapeamentos deste mapa.
- compute: default V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction), tenta calcular um mapeamento para a chave especificada e seu valor mapeado atual (ou nulo se não houver mapeamento atual).
- computeIfAbsent: default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction), se a chave especificada ainda não estiver associada a um valor (ou estiver mapeada para nulo), tenta calcular seu valor usando a função de mapeamento fornecida e insere-o neste mapa, a menos que seja nulo.
- computeIfPresent: default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction), se o valor da chave especificada estiver presente e não for nulo, tenta calcular um novo mapeamento dada a chave e seu valor mapeado atual.
- containsKey: boolean containsKey(Object key), retornará true se este mapa contiver um mapeamento para a chave especificada.
- containsValue: boolean containsValue(Object value), retornará true se este mapa mapear uma ou mais chaves para o valor especificado.
- entrySet: Set<Map.Entry<K,V>> entrySet(), retorna uma visualização Set dos mapeamentos contidos neste mapa.
- equals: boolean equals(Object o), compara o objeto especificado com este mapa quanto à igualdade.

- forEach: default void forEach(BiConsumer<? super K,? super V> action), executa a ação especificada para cada entrada neste mapa até que todas as entradas tenham sido processadas ou a ação gere uma exceção.

- get: V get(Object key), retorna o valor para o qual a chave especificada está mapeada ou nulo se este mapa não contiver nenhum mapeamento para a chave.

- getOrDefault: default V getOrDefault(Object key, V defaultValue), retorna o valor para o qual a chave especificada está mapeada ou defaultValue se este mapa não contiver nenhum mapeamento para a chave.

- hashCode: int hashCode(), retorna o hash valor do código hash para este mapa.

- isEmpity: boolean isEmpity(), retornará true se este mapa não contiver mapeamentos de valores-chave.

- keySet: Set<K> keySet(), retorna uma visualização Set das chaves contidas neste mapa.

- merge: default B merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction), Se a chave especificada ainda não estiver associada a um valor ou estiver associada a nulo, associa-a ao valor não nulo fornecido.

- put: V put(K key, V value), associa o valor especificado à chave especificada neste mapa.

- putAll: void putAll(Map<? extends K,? extends V> m), copies all of the mappings from the specified map to this map.

- putIfAbsent: default V putIfAbsent(K key, V value), se a chave especificada ainda não estiver associada a um valor (ou estiver mapeada para nulo), associa-a ao valor fornecido e retorna nulo, caso contrário, retorna o valor atual.

- remove: V remove(Object key), remove o mapeamento de uma chave deste mapa, se estiver presente.

- Remove: default boolean remove(Object key, Object value), remove a entrada da chave especificada somente se ela estiver atualmente mapeada para o valor especificado.

- replace: default V replace(K key, V value), substitui a entrada da chave especificada somente se ela estiver atualmente mapeada para algum valor.

- replace: default boolean replace(K key, V oldValue, V newValue), substitui a entrada da chave especificada somente se estiver atualmente mapeada para o valor especificado.

- replaceAll: default void replaceAll(BiFunction<? super K,? super V,? extends V> function), rubstitui o valor de cada entrada pelo resultado da invocação da função especificada nessa entrada até que todas as entradas tenham sido processadas ou a função gere uma exceção.

- size: int size(), retorna o número de mapeamentos de valores-chave neste mapa.

- values: Collenction<V> values(), retorna uma visualização de coleção dos valores contidos neste mapa.

3. CLASSES

3.1. JAVA CLASSES

Classes em Java são os principais blocos de construção da linguagem e representam um conceito fundamental da programação orientada a objetos (POO). Elas são usadas para criar objetos, que são instâncias dessas classes, e organizam o código em estruturas lógicas e reutilizáveis.
Suas principais características sao

3.1.1. Definição De Estrutura:

Uma classe em Java serve como um modelo ou blueprint para criar objetos. Ela define a estrutura de um objeto, especificando seus campos (variáveis de instância) e métodos (funções).

3.1.2. Encapsulamento

As classes permitem encapsular dados e comportamentos relacionados em um único pacote. Os campos de uma classe geralmente são definidos como privados ou protegidos, o que significa que só podem ser acessados dentro da própria classe ou de suas subclasses. Isso promove a segurança dos dados e a coesão do código.

3.1.3. Reutilização de Código

Classes são usadas para promover a reutilização de código. Você pode criar uma classe com funcionalidades específicas e usá-la em várias partes do seu programa.

3.1.4. Herança

Classes em Java podem ser usadas para criar hierarquias de classes. Uma classe pode herdar campos e métodos de outra classe (classe base ou superclasse), permitindo a criação de subclasses que estendem ou especializam o comportamento da classe base.

3.1.5. Polimorfismo

O polimorfismo é uma característica fundamental da POO, e as classes em Java desempenham um papel essencial nisso. Ele permite que objetos de subclasses sejam tratados como objetos de superclasses, facilitando a criação de código flexível e genérico.

3.1.6. Instanciação de Objetos

Uma classe é como um plano de construção para objetos. Para criar um objeto a partir de uma classe, você instancia a classe usando o operador `new`. Cada objeto instanciado a partir de uma classe possui seu próprio conjunto de campos, mas compartilha os métodos definidos na classe.

3.1.7. Métodos

As classes definem métodos que especificam o comportamento dos objetos. Métodos podem executar operações, acessar e modificar os campos da classe ou realizar qualquer ação desejada.

3.1.8. Construtores

As classes podem ter construtores, que são métodos especiais usados para inicializar objetos no momento da criação. Um construtor é chamado quando você cria um novo objeto da classe.

3.1.9. Modificadores de Acesso

As classes em Java podem usar modificadores de acesso, como `public`, `private`, `protected` e `default`, para controlar a visibilidade de campos e métodos, garantindo o encapsulamento e a segurança dos dados.

3.1.10. Implementação de Interfaces

As classes podem implementar interfaces, o que significa que elas concordam em fornecer implementações para os métodos definidos nas interfaces. Isso permite que as classes adotem um contrato de comportamento específico.

Classes são a base da organização e estruturação de código em Java e desempenham um papel fundamental na criação de programas orientados a objetos. Elas ajudam a promover a reutilização de código, a coesão, a modularidade e a manutenção de software de alta qualidade.

3.2 ABSTRACTSET

A classe AbstractSet é uma classe abstrata que fornece uma implementação base para a interface Set. Ela lida com operações comuns para conjuntos, como adicionar e verificar a existência de elementos.

3.2.1. Vantagens

- Fácil de usar.
- Não permite elementos duplicados.

3.2.2. Desvantagens

- Não permite elementos duplicados.

3.2.3. Exemplos de Aplicação

Armazenamento de dados sem duplicatas.

Um exemplo usando um conjunto para armazenar uma lista de cores exclusivas em um aplicativo de design:

```
Set<String> coresUnicas = new HashSet<>();
coresUnicas.add("Vermelho");
coresUnicas.add("Verde");
coresUnicas.add("Azul");
```

3.3. ABSTRACTLIST

A classe AbstractList é uma subclasse abstrata de AbstractCollection que fornece uma implementação base para a interface List. Ela lida com operações comuns para listas ordenadas, como adicionar, remover e acessar elementos por índice.

3.3.1. Vantagens

- Fácil de usar.
- Permite elementos duplicados.
- Permite acesso aleatório aos elementos.

3.3.2. Desvantagens

- Acesso aleatório pode ser lento em listas grandes.

3.3.3. Exemplos de Aplicação

Armazenamento de dados em ordem específica.

Um exemplo usando uma lista para armazenar tarefas em uma lista de afazeres:

```
List<String> afazeres = new ArrayList<>();
afazeres.add("Comprar mantimentos");
afazeres.add("Ir ao médico");
afazeres.add("Estudar para o exame”);
```

3.4 HASHSET

A classe HashSet implementa a interface Set usando uma tabela hash para armazenar os elementos. Ela permite armazenar elementos exclusivos e é eficiente para verificar a presença de um elemento.

3.4.1 Vantagens

- Rápido para adicionar e remover elementos.
- Não permite elementos duplicados.

3.4.2 Desvantagens

- Não permite acesso aleatório aos elementos.
- Não mantém a ordem dos elementos inseridos.

3.4.3 Exemplos de Aplicação

Armazenamento de dados sem duplicatas em ordem não específica.

Um exemplo usando usando um conjunto para armazenar IDs de produtos exclusivos em um carrinho de compras online:

```
Set<Integer> carrinhoDeCompras = new HashSet<>();
carrinhoDeCompras.add(101);
carrinhoDeCompras.add(205);
carrinhoDeCompras.add(101); // Não adicionará novamente
```

3.5. LINKEDLIST

A classe LinkedList implementa a interface List usando uma lista duplamente encadeada para armazenar os elementos. É eficiente para inserções e remoções no início, no final ou em qualquer posição da lista.

3.5.1. Vantagens

- Permite acesso aleatório aos elementos.
- Fácil de adicionar e remover elementos no início ou no final da lista.

3.5.2. Desvantagens

- Acesso aleatório pode ser lento em listas grandes.
- Adicionar ou remover elementos no meio da lista pode ser lento em listas grandes.

3.5.3. Exemplos de Aplicação

Armazenamento de dados em ordem específica com fácil adição/remoção no início/fim da lista.

Um exemplo usando uma lista encadeada para implementar uma fila de espera em um sistema de atendimento ao cliente.

```
LinkedList<String> filaDeEspera = new LinkedList<>();
filaDeEspera.addLast("Cliente A");
filaDeEspera.addLast("Cliente B");
filaDeEspera.addFirst("Cliente VIP"); // Adiciona na frente da fila
```

3.6. VECTOR

IA classe Vector implementa a interface List usando um array dinâmico para armazenar os elementos. É semelhante a ArrayList e é sincronizado, tornando-o seguro para uso em ambientes multi-thread.

3.6.1. Vantagens

- Permite acesso aleatório aos elementos.
- Fácil de adicionar e remover elementos no final da lista.

3.6.2. Desvantagens

- Acesso aleatório pode ser lento em listas grandes.
- Adicionar ou remover elementos no meio da lista pode ser lento em listas grandes.

3.6.3. Exemplos de Aplicação

Armazenamento de dados em ordem específica com fácil adição/remoção no final da lista.

Um exemplo usando um vetor para armazenar a lista de pedidos de um restaurante:

```
Vector<String> pedidos = new Vector<>();
pedidos.add("Pizza Margherita");
pedidos.add("Hambúrguer com batatas fritas");
pedidos.add("Sorvete de chocolate");
```

3.7. ARRAYLIST

Implementa a interface List usando um array dinâmico para armazenar os elementos.

3.7.1. Vantagens

- Permite acesso aleatório aos elementos.
- Fácil de adicionar e remover elementos no final da lista.

3.7.2. Desvantagens

- Acesso aleatório pode ser lento em listas grandes.
- Adicionar ou remover elementos no meio da lista pode ser lento em listas grandes.

3.7.3. Exemplos de Aplicação

Armazenamento de dados em ordem específica com fácil adição/remoção no final da lista.

Um exemplo Usando um ArrayList para armazenar informações de alunos em uma lista de presença:

```
ArrayList<String> listaDePresenca = new ArrayList<>();
listaDePresenca.add("Alice");
listaDePresenca.add("Bob");
listaDePresenca.add("Carol");
```

3.8. DICTIONARY

A classe Dictionary é uma classe legada que define um dicionário, que mapeia chaves exclusivas para valores exclusivos.

3.8.1. Vantagens

- Oferece mapeamento de chave-valor exclusivo.

3.8.2. Desvantagens

- É uma classe antiga e não é amplamente usada nas aplicações Java modernas.
- Não permite valores nulos ou chaves duplicadas.

3.8.3. Exemplos de Aplicação

Armazenamento de dados com chaves exclusivas e valores exclusivos.

Um exemplo simples de uso de Dictionary para mapear cores para seus códigos em um aplicativo de design.

```
Dictionary<String, String> cores = new Hashtable<>();
cores.put("Vermelho", "#FF0000");
cores.put("Verde", "#00FF00");

String codigoVermelho = cores.get("Vermelho");
```

3.9. ABSTRACTMAP

A classe AbstractMap é uma classe abstrata que implementa a interface Map e fornece uma implementação padrão para métodos comuns.

3.9.1. Vantagens

- Fornece uma base sólida para a implementação de classes Map personalizadas.

3.9.2. Desvantagens

- A classe em si não é instanciável, é destinada a ser estendida por outras classes.

3.9.3. Exemplos de Aplicação

A classe AbstractMap não é usada diretamente, mas serve como base para implementações concretas de Map

Exemplo de uma implementação personalizada de AbstractMap para mapear nomes de países para suas capitais:

```
class MeuMapaDeCapitais extends AbstractMap<String, String> {
    private Set<Entry<String, String>> entrySet = new HashSet<>();

    @Override
    public Set<Entry<String, String>> entrySet() {
        return entrySet;
    }
}

MeuMapaDeCapitais mapaCapitais = new MeuMapaDeCapitais();
mapaCapitais.put("Brasil", "Brasília");
mapaCapitais.put("EUA", "Washington, D.C.");

String capitalDoBrasil = mapaCapitais.get(“Brasil");
```

3.10. HASHTABLE

Implementa a interface Map usando uma tabela hash para armazenar os pares chave-valor.

3.10.1. Vantagens

- Rápido para adicionar e remover pares chave-valor.

3.10.2. Desvantagens

- Não permite chaves ou valores nulos.

3.10.3. Exemplos de Aplicação

Armazenamento de dados com pares chave-valor sem duplicatas em ordem não específica.

Exemplo Usando uma Hashtable para manter uma lista de contas de usuário com suas senhas em um sistema de autenticação:

```
Hashtable<String, String> contasDeUsuario = new Hashtable<>();
contasDeUsuario.put("alice", "senha123");
contasDeUsuario.put("bob", “senha456");

String senhaDoBob = contasDeUsuario.get("bob");
```

3.11. HASHMAP

Implementa a interface Map usando uma tabela hash para armazenar os pares chave-valor.

3.11.1. Vantagens

- Rápido para adicionar e remover pares chave-valor.

3.11.2. Desvantagens

- Não permite chaves ou valores nulos.

3.11.3. Exemplos de Aplicação

Armazenamento de dados com pares chave-valor sem duplicatas em ordem não específica.

```
HashMap<String, String> capitais = new HashMap<>();
capitais.put("Brasil", "Brasília");
capitais.put("EUA", "Washington, D.C.");
```

REFERÊNCIAS

1 **Java collections framework.** Wikipedia, 2010. Disponivel em: https://en.wikipedia.org/wiki/Java_collections_framework. Acesso em: 31 out. 2023.

2 **Java.util.Collection.** docs.oracle, 1993. Disponivel em: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html. Acesso em: 02 nov. 2023.

3 **Java.util.Iterable.** docs.oracle, 1993. Disponivel em: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html. Acesso em: 02 nov. 2023.

4 **Java.util.Map.** docs.oracle, 1993. Disponivel em: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html. Acesso em: 02 nov. 2023.

5 **Java.util.Set.** docs.oracle, 1993. Disponivel em: https://docs.oracle.com/javase/8/docs/api/java/util/Set.html. Acesso em: 02 nov. 2023.

6 **Java.util.List.** docs.oracle, 1993. Disponivel em: https://docs.oracle.com/javase/8/docs/api/java/util/List.html. Acesso em: 02 nov. 2023.

7 **Java.util.Queue.** docs.oracle, 1993. Disponivel em: https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html. Acesso em: 02 nov. 2023.

8 Claudio. **Quando usar Java Collections – Parte 1.** DevMedia, 2007. Disponivel em: https://www.devmedia.com.br/quando-usar-java-collections-parte-i/5025. Acesso em: 02 nov. 2023.

9 **Set in Java.** geeksforgeeks, s.d. Disponivel em: https://www.geeksforgeeks.org/set-in-java/. Acesso em: 02 nov. 2023.

10. https://stackoverflow.com/questions/44614463/what-is-the-main-difference-between-linkedlist-hashset-and-hashmap

11. https://tutoriais.edu.lat/pub/java/java-collections/java-estrutura-de-colecoes

12. https://stackoverflow.com/questions/21220873/abstractlist-and-list-in-java

13. https://www.w3schools.blog/hashset-hashmap-hashtable-java

14. https://stackoverflow.com/questions/24165406/do-arraylist-or-hashset-classes-implements-collection-interface-implicitly

15. https://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html
16. https://www.geeksforgeeks.org/abstractset-class-in-java-with-examples/
17. https://docs.oracle.com/javase/8/docs/api/java/util/AbstractList.html
18. https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html
19. https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
20. https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html
21. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
22. https://docs.oracle.com/javase/8/docs/api/java/util/Dictionary.html
23. https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html
24. https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html
25. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html