

# Collections Framework

Programação Orientada a Objetos — QXD0007



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz  
gomes.atilio@ufc.br

Universidade Federal do Ceará

2º semestre/2021



# Leitura para esta aula

- **Capítulo 16** (Coleções Genéricas) do livro Java Como Programar, Décima Edição.

# Objetivos

- Interface List
- Interface Iterator
- Interface ListIterator
- Classe ArrayList
- Classe LinkedList
- Classe Collections e seus métodos
- Interface Set
- Classe HashSet
- Classe TreeSet

# Introdução



# Collections Framework

- **Collections framework:** Java contém classes e interfaces que poupam trabalho ao programador e oferecem mecanismos para agrupar e processar objetos em conjuntos.
- Conjuntos de objetos são chamados de **coleções**.

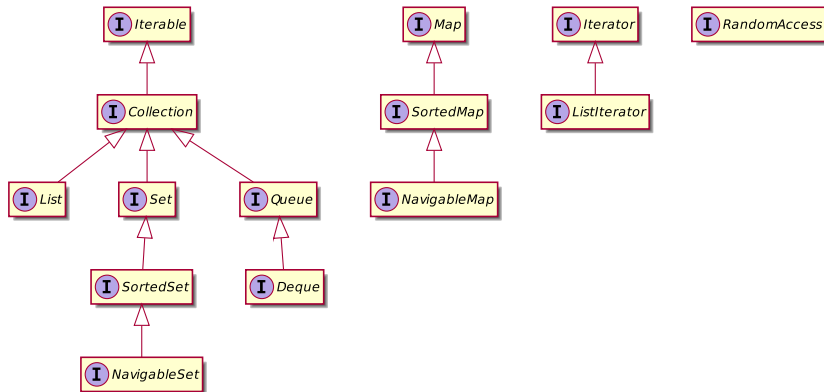
# Collections Framework

- **Collections framework:** Java contém classes e interfaces que poupam trabalho ao programador e oferecem mecanismos para agrupar e processar objetos em conjuntos.
- Conjuntos de objetos são chamados de **coleções**.
- Algumas classes que implementam coleções de objetos podem ter funções semelhantes mas implementações internamente diferentes (por exemplo: lista encadeada  $\times$  vetor).

# Collections Framework

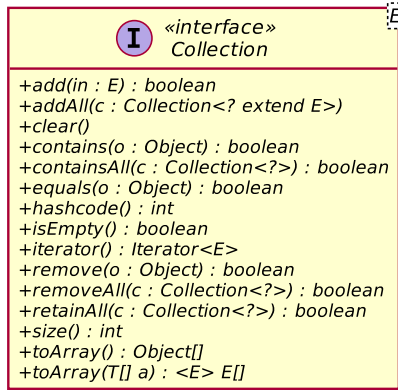
- **Collections framework:** Java contém classes e interfaces que poupam trabalho ao programador e oferecem mecanismos para agrupar e processar objetos em conjuntos.
- Conjuntos de objetos são chamados de **coleções**.
- Algumas classes que implementam coleções de objetos podem ter funções semelhantes mas implementações internamente diferentes (por exemplo: lista encadeada  $\times$  vetor).
- Por esse motivo, Java separa a representação da coleção em **interfaces** e **implementações**. O programador pode, então, para uma dada estrutura de dado ou coleção, escolher a implementação mais adequada à sua necessidade.

# Interfaces da *collections framework*

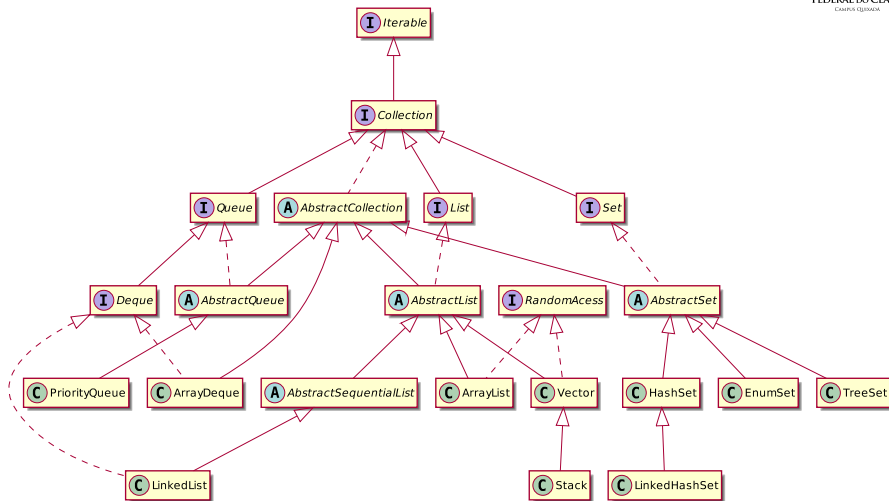




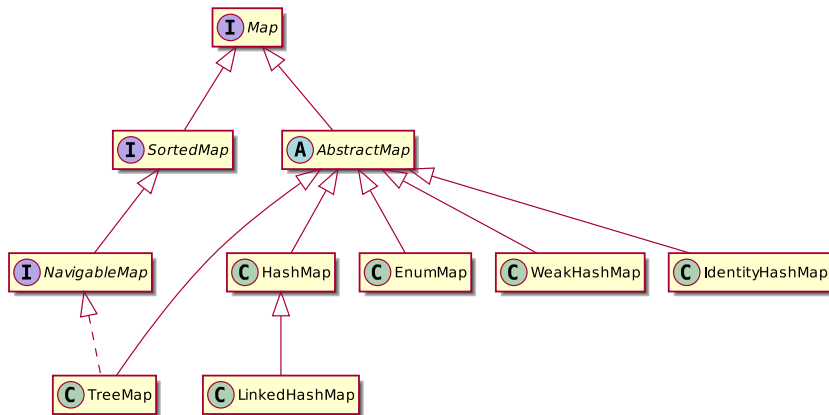
# Interface Collection



API Java: <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collection.html>



Hierarquia de classes — List, Set e Queue



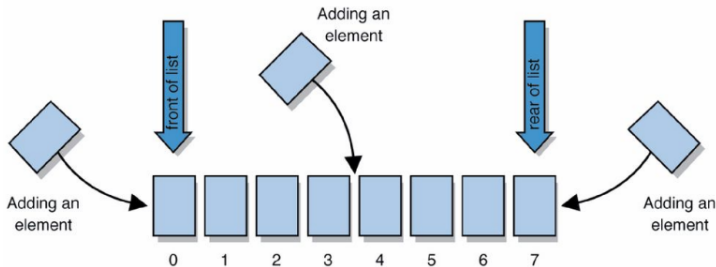
Hierarquia de classes — Map

# Listas



# Listas

- Uma **lista** é uma coleção de objetos que permite elementos duplicados e mantém uma ordenação específica entre os elementos.



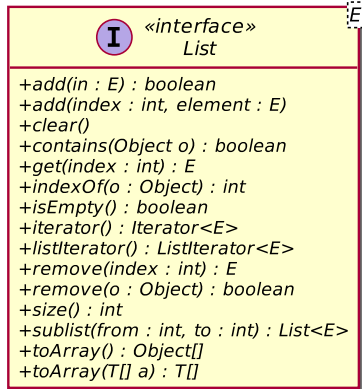
# Interface List<E>

- `java.util.List`: interface que especifica os métodos que uma classe deve ter para ser uma lista.
  - Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.

# Interface List<E>

- `java.util.List`: interface que especifica os métodos que uma classe deve ter para ser uma lista.
  - Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.
- `List` estende a interface `Collection` e declara o comportamento de uma coleção que armazena uma sequência de elementos.
  - Elementos podem ser inseridos ou acessados por sua posição na lista, usando um índice começando do zero.
  - Uma lista pode conter elementos duplicados.
  - Além dos métodos definidos por `Collection`, `List` define seus próprios métodos.

# Interface List&ltE>



API Java: [https://download.java.net/java/early\\_access/panama/docs/api/java.base/java/util/List.html](https://download.java.net/java/early_access/panama/docs/api/java.base/java/util/List.html)



## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista
- `void clear()`: remove todos os elementos da lista

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista
- `void clear()`: remove todos os elementos da lista

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista
- `void clear()`: remove todos os elementos da lista
- `E remove(int index)`: remove o elemento na posição especificada

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista
- `void clear()`: remove todos os elementos da lista
- `E remove(int index)`: remove o elemento na posição especificada
- `boolean remove(Object o)`: remove a primeira ocorrência do elemento



## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista
- `void clear()`: remove todos os elementos da lista
- `E remove(int index)`: remove o elemento na posição especificada
- `boolean remove(Object o)`: remove a primeira ocorrência do elemento
- `E get(int index)`: retorna o elemento na posição especificada na lista.  
Lança `IndexOutOfBoundsException` se o índice estiver fora do intervalo.

## List<E> — Alguns Métodos

- `void add(E e)` insere o elemento no final da lista
- `void add(int index, E e)` insere o elemento na posição especificada
- `int size()`: devolve o número de elementos na lista
- `void clear()`: remove todos os elementos da lista
- `E remove(int index)`: remove o elemento na posição especificada
- `boolean remove(Object o)`: remove a primeira ocorrência do elemento
- `E get(int index)`: retorna o elemento na posição especificada na lista. Lança `IndexOutOfBoundsException` se o índice estiver fora do intervalo.
- `boolean isEmpty()`: devolve true se lista vazia

# List<E> — Mais detalhes

API Java: [https://download.java.net/java/early\\_access/panama/docs/api/java.base/java/util/List.html](https://download.java.net/java/early_access/panama/docs/api/java.base/java/util/List.html)

# Classe ArrayList<E>

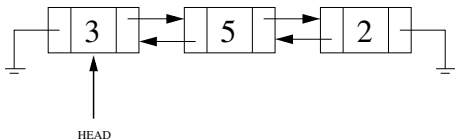
- `java.util.ArrayList`: classe que implementa a interface `List`.
  - Implementa `List` usando internamente um **array redimensionável**.
  - Quando precisa de mais espaço, um novo array de tamanho maior é alocado e o conteúdo do array antigo é copiado para o novo array. O array antigo é então liberado e o novo array toma o lugar do antigo.

# Classe ArrayList<E>

- `java.util.ArrayList`: classe que implementa a interface `List`.
  - Implementa `List` usando internamente um **array redimensionável**.
  - Quando precisa de mais espaço, um novo array de tamanho maior é alocado e o conteúdo do array antigo é copiado para o novo array. O array antigo é então liberado e o novo array toma o lugar do antigo.
- Operações de acesso a um elemento pelo índice são **rápidas**.
- Inserção e remoção no final do `ArrayList` são operações **rápidas**.
- Operações de inserção e remoção em outras posições são, em geral, **lentas**.

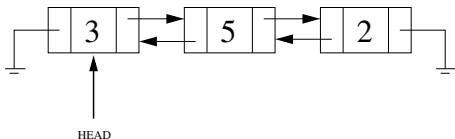
# Classe LinkedList<E>

- `java.util.LinkedList`: classe que implementa a interface `List`.
  - Implementa `List` usando internamente uma **lista duplamente encadeada**.
  - Quando precisa de mais espaço, cria um novo nó e adiciona à lista.



# Classe LinkedList<E>

- `java.util.LinkedList`: classe que implementa a interface `List`.
  - Implementa `List` usando internamente uma **lista duplamente encadeada**.
  - Quando precisa de mais espaço, cria um novo nó e adiciona à lista.



- Operações de acesso a um elemento pelo índice são **lentas**.
- Operações de inserção e remoção são **rápidas**.

# Exemplo

- Analisar o [Projeto1](#)
- analisar Projeto [RoboComMemoria](#)

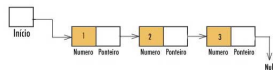
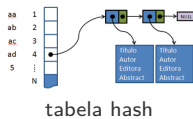
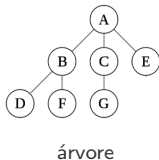
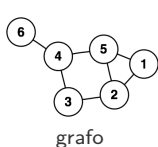


# Interface Iterator



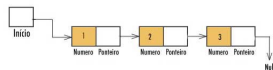
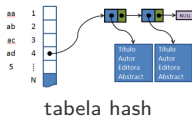
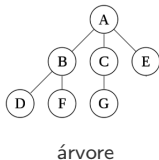
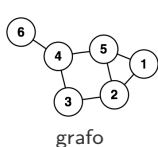
# Um problema de Engenharia de Software

- Existem diversas estruturas de dados na computação, algumas são baseadas em listas lineares, mas outras podem ter como base uma árvore, um grafo, ou até mesmo uma tabela hash.



# Um problema de Engenharia de Software

- Existem diversas estruturas de dados na computação, algumas são baseadas em listas lineares, mas outras podem ter como base uma árvore, um grafo, ou até mesmo uma tabela hash.



**Pergunta:** é possível criar um mecanismo comum a todas essas estruturas que possibilite percorrer todos os seus elementos?

# Padrão de Projeto Iterator

- O **Iterator** é um padrão de projeto comportamental que permite a passagem sequencial através de uma estrutura de dados complexa sem expor seus detalhes internos.
- Graças ao Iterator, os clientes podem examinar elementos de diferentes coleções de maneira semelhante usando uma única interface iterador.

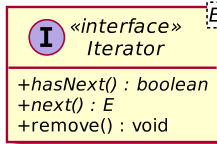
## List<E> — Mais Métodos

A interface `List` possui três métodos que retornam objetos que nos permitem percorrer todos os elementos armazenados na coleção de forma sistemática.

Um dos métodos retorna um objeto do tipo `Iterator` e dois outros retornam um objeto do tipo `ListIterator`.

- `Iterator<E> iterator()`: retorna um iterador do tipo `Iterator<E>`, para que possamos percorrer a lista sequencialmente.
- `ListIterator<E> listIterator()`: retorna um iterador do tipo `ListIterator<E>`, para que possamos percorrer a lista sequencialmente.
- `ListIterator<E> listIterator(int index)` retorna um iterador do tipo `ListIterator<E>`, começando da posição especificada no parâmetro `index`.

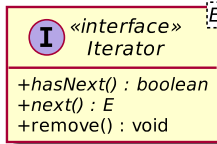
# Interface Iterator



## Alguns Métodos:

- `boolean hasNext()`: retorna `true`, se ainda houver elementos na coleção a serem processados.

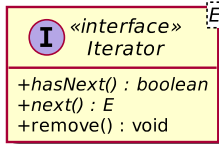
# Interface Iterator



## Alguns Métodos:

- `boolean hasNext()`: retorna true, se ainda houver elementos na coleção a serem processados.
- `E next()`: retorna o próximo elemento da coleção. Se não houver próximo elemento, lança uma `NoSuchElementException`.

# Interface Iterator

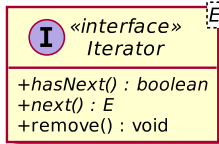


## Alguns Métodos:

- `boolean hasNext()`: retorna `true`, se ainda houver elementos na coleção a serem processados.
- `E next()`: retorna o próximo elemento da coleção. Se não houver próximo elemento, lança uma `NoSuchElementException`.
- `remove()`: ao invocar este método, ele remove da lista original o último elemento retornado por `next()`.



# Interface Iterator



## Alguns Métodos:

- `boolean hasNext()`: retorna `true`, se ainda houver elementos na coleção a serem processados.
- `E next()`: retorna o próximo elemento da coleção. Se não houver próximo elemento, lança uma `NoSuchElementException`.
- `remove()`: ao invocar este método, ele remove da lista original o último elemento retornado por `next()`.
  - lança uma `IllegalStateException` se o método `next()` nunca tiver sido chamado ou se `remove` tiver sido chamado mais de uma vez depois da última chamada a `next()`.

# Interface Iterator

Em geral, a fim de usar um iterador para percorrer o conteúdo de uma coleção, seguimos siga estas etapas:

- Obtemos um iterador para o início da coleção chamando o método `iterator()` da coleção.
- Criamos um loop que faz uma chamada para `hasNext()` e repete até que esse método retorne `true`.
- Dentro do loop, acessamos o elemento chamando `next()`.

# Interface Iterator – Exemplo

```
1 public class Exemplo01 {
2     public static void main(String[] args) {
3         List<String> lista = new LinkedList<>();
4         lista.add("A");
5         lista.add("B");
6         lista.add("C");
7
8         // usa um iterador para mostrar o conteúdo do array
9         System.out.println("Array original: " + lista);
10
11         Iterator<String> it = lista.iterator();
12
13         while(it.hasNext()) {
14             if(it.next().equals("B"))
15                 it.remove();
16         }
17
18         System.out.println("Array modificado: " + lista);
19     }
20 }
```

# Interface Iterator

- Analisar arquivo `CollectionTest.java`

## O loop `for each` ou `enhanced-for`

- O loop `for each` itera sobre qualquer objeto que implemente a interface `Iterable`, que é uma interface que possui um único método abstrato:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
    . . .
}
```

## O loop `for each` ou `enhanced-for`

- O loop `for each` itera sobre qualquer objeto que implemente a interface `Iterable`, que é uma interface que possui um único método abstrato:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
    . . .
}
```

- A interface `Collection` estende a interface `Iterable`. Portanto, podemos usar essa outra versão do loop `for` com qualquer coleção da biblioteca padrão.

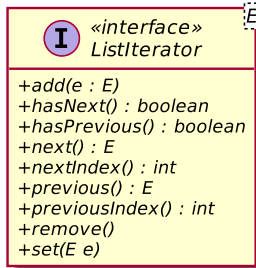
## O loop `for each` ou `enhanced-for`

- O loop `for each` itera sobre qualquer objeto que implemente a interface `Iterable`, que é uma interface que possui um único método abstrato:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
    . . .
}
```

- A interface `Collection` estende a interface `Iterable`. Portanto, podemos usar essa outra versão do loop `for` com qualquer coleção da biblioteca padrão.
- Pela sua simplicidade, muitas vezes é preferível usar o `for each` a usar iteradores explicitamente. Por outro lado, se além de percorrer a coleção, quisermos remover elementos, podemos também o método `remove()` do iterador.

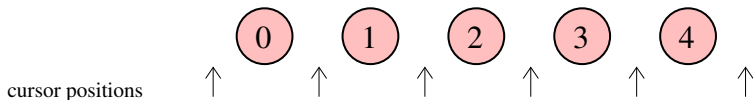
# Interface ListIterator



- Um iterador para listas que permite ao programador percorrer a lista em qualquer direção, modificar a lista durante a iteração e obter a posição atual do iterador na lista.



# Interface ListIterator



- Um `ListIterator` não aponta para um elemento.
- A posição do cursor sempre fica entre o elemento que seria retornado por uma chamada para `previous()` e o elemento que seria retornado por uma chamada para `next()`.
- Considerando que a lista tem  $n$  elementos, o `ListIterator` possui  $n + 1$  posições de cursor possíveis, conforme ilustrado pelas  $\uparrow$  na figura acima.

# Interface ListIterator

## Alguns Métodos:

- `boolean hasNext()`: Retorna true se este iterador de lista tiver mais elementos ao percorrer a lista na direção para frente.
- `boolean hasPrevious()`: Retorna true se este iterador de lista tiver mais elementos ao percorrer a lista na direção para trás.
- `E next()`: retorna o próximo elemento da coleção e avança o cursor uma posição para frente.
- `E previous()`: retorna o elemento anterior e move o cursor uma posição para trás.
- `void add(E e)`: insere o elemento na lista. O elemento é inserido antes do elemento que seria retornado por `next()`.
- `remove()`: remove da lista o último elemento que foi retornado por `next()` ou `previous()`.

# Interface ListIterator

- Analisar arquivo `ListTest.java`

# Classe Collections



# Classe `java.util.Collections`

- A classe `Collections` é composta de **métodos estáticos** que, na sua maioria, implementam algoritmos clássicos de alta performance.
- Os métodos desta classe são **polimórficos**, ou seja, eles podem operar sobre objetos que implementam interfaces específicas, independentemente das implementações subjacentes.

# Método `Collections.sort`

- `Collections` possui o método estático `sort`, que é usado para ordenar objetos que implementam a interface `List`.

# Método `Collections.sort`

- `Collections` possui o método estático `sort`, que é usado para ordenar objetos que implementam a interface `List`.
- Há duas versões deste método:
  - `void sort(List<T> list)`  
recebe um `List` como argumento e o ordena por ordem crescente se os seus elementos pertencerem a uma classe que implemente a interface `Comparable<T>`.
  - `void sort(List<T> list, Comparator<T> c)`  
recebe um `List` como argumento e o ordena de acordo com a ordem especificada pelo objeto do tipo `Comparator<T>`.

# Collections.sort — Exemplo 1

```
1 import java.util.List;
2 import java.util.Arrays;
3 import java.util.Collections;
4
5 public class Sort1 {
6     public static void main(String[] args) {
7         String[] nomes = {"Célia", "Alberto", "Paula",
8                             "Bruna", "Júlia", "Hugo"};
9
10        // cria uma lista
11        List<String> lista = Arrays.asList(nomes);
12
13        System.out.println("Lista original: " + lista);
14
15        Collections.sort(lista); // ordena a lista
16
17        System.out.println("Lista ordenada: " + lista);
18    }
19 }
```

- Como a classe `String` implementa a interface `Comparable`, então a lista será ordenada corretamente.



# Collections.sort

- **Problema:** E se quiséssemos ordenar a lista de Strings do exemplo anterior não na sua ordem natural (ordem crescente), mas sim em ordem decrescente? Como fazer?

# Collections.sort

- **Problema:** E se quiséssemos ordenar a lista de Strings do exemplo anterior não na sua ordem natural (ordem crescente), mas sim em ordem decrescente? Como fazer?
- **Dificuldades:** Não podemos mexer no código da classe String e mudar a implementação do método `compareTo`. Por consequência, não podemos usar o método `sort` que utilizamos no exemplo anterior.

# Collections.sort

- **Problema:** E se quiséssemos ordenar a lista de Strings do exemplo anterior não na sua ordem natural (ordem crescente), mas sim em ordem decrescente? Como fazer?
- **Dificuldades:** Não podemos mexer no código da classe String e mudar a implementação do método `compareTo`. Por consequência, não podemos usar o método `sort` que utilizamos no exemplo anterior.
- **Solução 1:** Porém, a classe Collections possui uma versão sobrecarregada do método `sort` que recebe um List e um Comparator. Logo, basta escrever uma classe que implemente a interface Comparator e implemente a nova lógica de ordenação.

# Collections.sort

- **Problema:** E se quiséssemos ordenar a lista de Strings do exemplo anterior não na sua ordem natural (ordem crescente), mas sim em ordem decrescente? Como fazer?
- **Dificuldades:** Não podemos mexer no código da classe String e mudar a implementação do método `compareTo`. Por consequência, não podemos usar o método `sort` que utilizamos no exemplo anterior.
- **Solução 1:** Porém, a classe Collections possui uma versão sobrecarregada do método `sort` que recebe um List e um Comparator. Logo, basta escrever uma classe que implemente a interface Comparator e implemente a nova lógica de ordenação.
- **Solução 2:** Para o caso específico de obter a ordem inversa da ordenação natural, a própria classe Collections já possui um método chamado `reverseOrder()` que retorna um objeto Comparator que força essa ordem reversa.

## Collections.sort — Exemplo 2

```
1 import java.util.List;
2 import java.util.Arrays;
3 import java.util.Collections;
4
5 public class Sort2 {
6     public static void main(String[] args) {
7         String[] nomes = {"Célia", "Alberto", "Paula",
8                             "Bruna", "Júlia", "Hugo"};
9
10        // cria uma lista
11        List<String> lista = Arrays.asList(nomes);
12
13        System.out.println("Lista original: " + lista);
14
15        Collections.sort(lista, Collections.reverseOrder());
16        System.out.print("Lista em ordem decrescente: ");
17        System.out.println(lista);
18    }
19 }
```

# Método Collections.reverse

```
1 import java.util.List;
2 import java.util.Arrays;
3 import java.util.Collections;
4
5 public class Reverse {
6     public static void main(String[] args) {
7         String[] nomes = {"Célia", "Alberto", "Paula",
8                             "Bruna", "Júlia", "Hugo"};
9
10        // cria uma lista
11        List<String> lista = Arrays.asList(nomes);
12
13        System.out.println("Lista original: " + lista);
14
15        Collections.reverse(lista); // inverte a lista
16
17        System.out.println("Lista invertida: " + lista);
18    }
19 }
```

# Método `Collections.shuffle`

- A classe `Collections` também fornece o método `shuffle`, que embaralha uma lista.
- Essa funcionalidade pode ser útil para implementação de jogos, simulações e testes.

# Método Collections.shuffle

```
1 import java.util.List;
2 import java.util.Arrays;
3 import java.util.Collections;
4
5 public class Shuffle {
6     public static void main(String[] args) {
7         Character[] letras = {'a','b','c','d','e','f','g',
8                               'h','i','j','k','l','m','n'};
9
10        // cria uma lista
11        List<Character> lista = Arrays.asList(letras);
12
13        System.out.println("Lista original: " + lista);
14
15        Collections.shuffle(lista); // embaralha
16
17        System.out.println("Lista embaralhada: " + lista);
18    }
19 }
```



## Outros métodos que operam em Lists

- `void fill(List<T> list, T obj)`: Substitui todos os elementos da lista com o elemento especificado. Todos os elementos da lista passam a referenciar o mesmo objeto.
- `void copy(List<T> dest, List<T> source)`: Copia todos os elementos da lista `source` na lista `dest`. A lista de destino deve ser maior ou igual a lista de origem, caso contrário, será lançada uma `IndexOutOfBoundsException`.
- `int binarySearch(List<T> list, T key)`: Este método busca um elemento específico na lista. Se o objeto for encontrado, o seu índice na lista é retornado; caso contrário, ele retorna um número negativo.
  - **Atenção:** Para que este algoritmo funcione corretamente, os elementos da lista devem estar ordenados por ordem crescente.

# Exemplos

- Analisar o arquivo `Algorithms1.java`
- Analisar o arquivo `BinarySearchTest.java`

# Conjuntos (Sets)



# Estrutura de dados: Conjunto

- **Conjuntos** são coleções de objetos que não permitem objetos duplicados.

Exemplos:

- (a) alunos de uma turma
- (b) números sorteados em uma loteria
- (c) letras do alfabeto

# Estrutura de dados: Conjunto

- **Conjuntos** são coleções de objetos que não permitem objetos duplicados.

Exemplos:

- (a) alunos de uma turma
  - (b) números sorteados em uma loteria
  - (c) letras do alfabeto
- Conjuntos podem ser vazios mas não podem ser infinitos pois sua representação seria impossível.
    - O tamanho dos conjuntos pode variar dinamicamente – não é necessário saber o tamanho do conjunto quando ele é criado.

# Estrutura de dados: Conjunto

- **Conjuntos** são coleções de objetos que não permitem objetos duplicados.

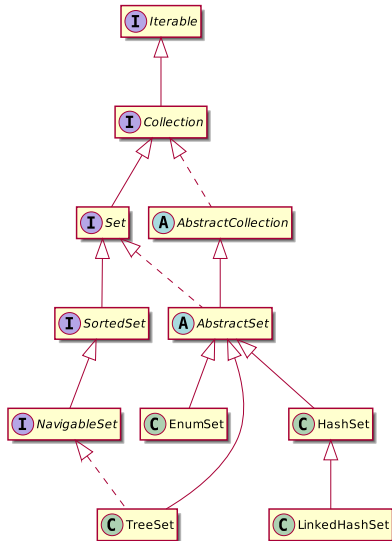
Exemplos:

- (a) alunos de uma turma
  - (b) números sorteados em uma loteria
  - (c) letras do alfabeto
- Conjuntos podem ser vazios mas não podem ser infinitos pois sua representação seria impossível.
    - O tamanho dos conjuntos pode variar dinamicamente – não é necessário saber o tamanho do conjunto quando ele é criado.
  - Outra característica fundamental dos conjuntos é que a ordem em que os objetos são armazenados no conjunto pode não ser a ordem na qual eles foram inseridos no conjunto.

# Estrutura de dados: Conjunto

- Uma outra diferença fundamental entre conjuntos e as outras estruturas de dados que vimos até agora é que não precisamos especificar índices para inserir um elemento em um conjunto.
- Em um conjunto, simplesmente inserimos um elemento sem explicitar posições.
- Logo, também não é possível acessar elementos usando índices.

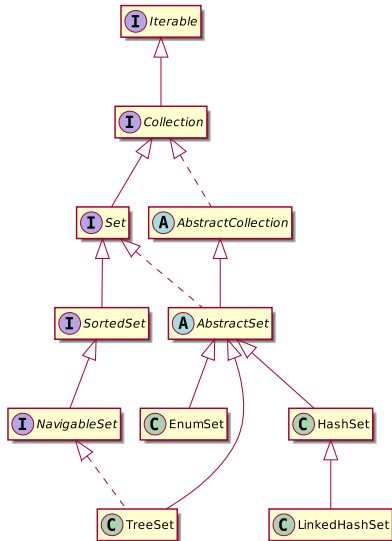
# Interface java.util.Set



- Em Java, a estrutura de dados **Conjunto** é definida pela interface **Set** e tem como suas principais implementações as classes **HashSet** e **TreeSet**.

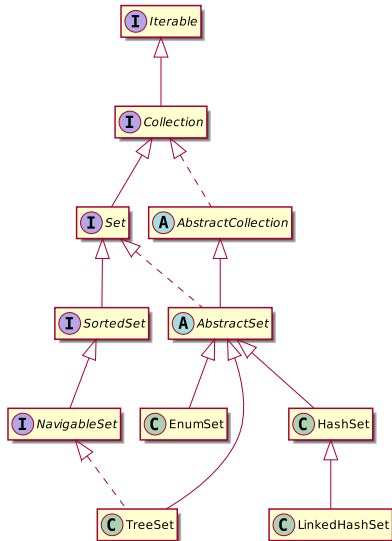


# Interface java.util.Set



- Em Java, a estrutura de dados **Conjunto** é definida pela interface **Set** e tem como suas principais implementações as classes **HashSet** e **TreeSet**.
- **Hash Set**: operações de inserção e remoção são rápidas. Porém, não estabelece nenhuma ordem em particular entre os objetos.

# Interface java.util.Set



- Em Java, a estrutura de dados **Conjunto** é definida pela interface **Set** e tem como suas principais implementações as classes **HashSet** e **TreeSet**.
- **Hash Set**: operações de inserção e remoção são rápidas. Porém, não estabelece nenhuma ordem em particular entre os objetos.
- **TreeSet**: preserva a ordem natural dos objetos, mas à custa de performance na inserção e remoção.

# Interface Set<E> – Métodos

- `boolean add(E e)`: Adiciona o elemento no conjunto se ele ainda não estiver presente. Se o elemento já estiver no conjunto, o conjunto fica inalterado e é retornado `false`.
- `boolean addAll(Collection<E> c)`: adiciona todos os elementos da coleção a este conjunto se eles ainda não estiverem no conjunto. Retorna `true` se este conjunto for modificado pela chamada deste método.
- `void clear()`: remove todos os elementos do conjunto.
- `boolean contains(Object o)`: retorna `true` se este conjunto contém o elemento especificado.
- `boolean containsAll(Collection<E> coll)`: retorna `true` se este conjunto contém todos os elementos da coleção especificada.

# Interface Set<E> – Métodos

- `boolean isEmpty()`: retorna `true` se o conjunto estiver vazio.
- `equals(Object o)`: Determina se este conjunto é igual ao objeto especificado.
- `Iterator<E> iterator()`: retorna um iterador para este conjunto.
- `boolean remove(Object o)`: remove do conjunto o elemento especificado se ele estiver presente.
- `boolean removeAll(Collection<E> c)`: remove deste conjunto todos os elementos que estão contidos na coleção especificada. Retorna `true` se este conjunto tiver sido modificado pela chamada deste método.

## Interface Set<E> – Métodos

- `boolean retainAll(Collection<E> c)`: realiza a operação de interseção entre conjuntos. Ou seja, permanecerão neste conjunto apenas os elementos que estiverem também contidos no conjunto especificado.
- `int size()`: retorna o número de elementos neste conjunto.
- `Object[] toArray()`: retorna um array de Object contendo todos os elementos neste conjunto. Modificações feitas no array retornado não refletem na estrutura interna deste conjunto.
- `T[] toArray(T[] a)`: retorna um array contendo todos os elementos deste conjunto. O tipo `T` do array deve ser um supertipo de todos os elementos contidos neste conjunto, caso contrário, uma exceção será lançada.

## Interface Set<E> – Método Estático

- `Set<E> copyOf(Collection<E> coll)`: retorna um conjunto **não-modificável** contendo os elementos da coleção especificada.
  - Elementos não podem ser adicionados ou modificados.
  - Tentar modificar o conjunto causará uma `UnsupportedOperationException`.

# Exemplo

- O usuário vai digitar uma sequência de palavras e gostaríamos de imprimir todas as palavras que foram digitadas, excluindo as repetições.

# Solução 1

```
1 import java.util.Set;
2 import java.util.HashSet;
3 import java.util.Scanner;
4
5 public class HashSetTest {
6     public static void main(String[] args) {
7         Scanner sc = new Scanner(System.in);
8         System.out.println("Digite as palavras: ");
9         String line = sc.nextLine();
10
11         // Construtor vazio (default)
12         Set<String> set = new HashSet<>();
13
14         sc = new Scanner(line);
15         while(sc.hasNext())
16             set.add(sc.next());
17
18         System.out.println("Palavras, sem duplicatas: ");
19         System.out.println(set);
20
21     }
22 }
```



## Solução 2

```
1 import java.util.Set;
2 import java.util.Arrays;
3 import java.util.HashSet;
4 import java.util.Scanner;
5
6 public class HashSetTest2 {
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9         System.out.println("Digite as palavras: ");
10        String line = sc.nextLine();
11
12        String[] a = line.split(" ");
13
14        // usando um dos construtores sobrecarregados
15        Set<String> set = new HashSet<>(Arrays.asList(a));
16
17        System.out.println("Palavras, sem duplicatas: ");
18        System.out.println(set);
19    }
20 }
```

# Classe HashSet



# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca

boca

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	NULL
1	→	NULL
2	→	NULL
3	→	NULL
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca

boca

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	NULL
1	→	NULL
2	→	NULL
3	→	NULL
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca

boca

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	NULL
1	→	NULL
2	→	NULL
3	→	NULL
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	NULL
1	→	NULL
2	→	NULL
3	→	NULL
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	NULL
1	→	NULL
2	→	NULL
3	→	broca
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	NULL
1	→	NULL
2	→	NULL
3	→	broca
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)



# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo

bela

bala

dia

escola

gratuito

ilha

0	→	boca
1	→	NULL
2	→	NULL
3	→	broca
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela

bala

dia

escola

gratuito

ilha

0	→	boca
1	→	NULL
2	→	NULL
3	→	broca
4	→	NULL
5	→	NULL
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela

bala

dia

escola

gratuito

ilha

0	→	boca
1	→	NULL
2	→	NULL
3	→	broca
4	→	NULL
5	→	<b>bolo</b>
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

bala

dia

escola

gratuito

ilha

0	→	boca
1	→	NULL
2	→	NULL
3	→	broca
4	→	NULL
5	→	bolo
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe `HashSet` é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

bala

dia

escola

gratuito

ilha

0	→	boca
1	→	NULL
2	→	bela
3	→	broca
4	→	NULL
5	→	bolo
6	→	NULL
7	→	NULL
8	→	NULL

- **Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia

escola

gratuito

ilha

0	→	boca
1	→	NULL
2	→	bela
3	→	broca
4	→	NULL
5	→	bolo
6	→	NULL
7	→	NULL
8	→	NULL

- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

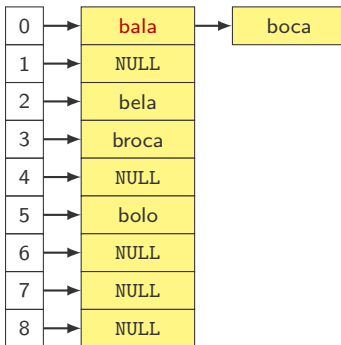
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia

escola

gratuito

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

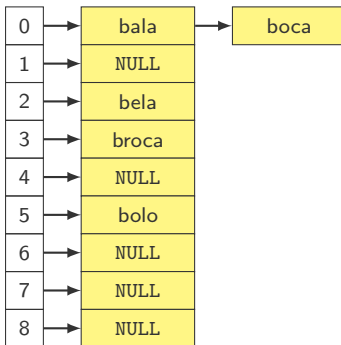
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola

gratuito

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)



# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

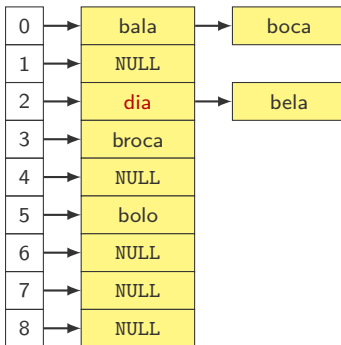
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola

gratuito

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

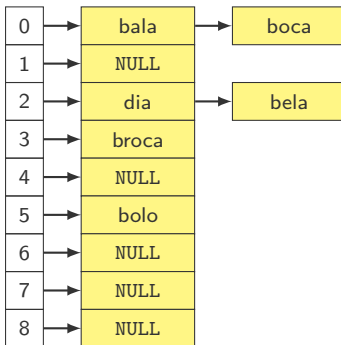
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola  $\rightsquigarrow h(\text{"escola"}) = 7$

gratuito

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

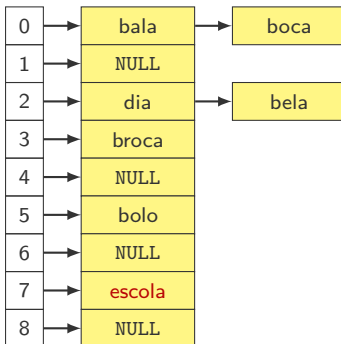
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola  $\rightsquigarrow h(\text{"escola"}) = 7$

gratuito

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

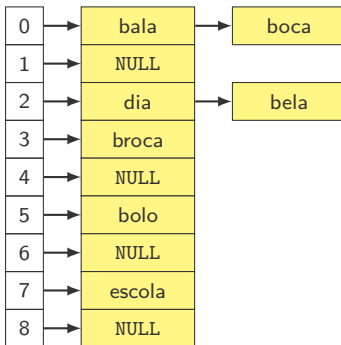
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola  $\rightsquigarrow h(\text{"escola"}) = 7$

gratuito  $\rightsquigarrow h(\text{"gratuito"}) = 0$

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

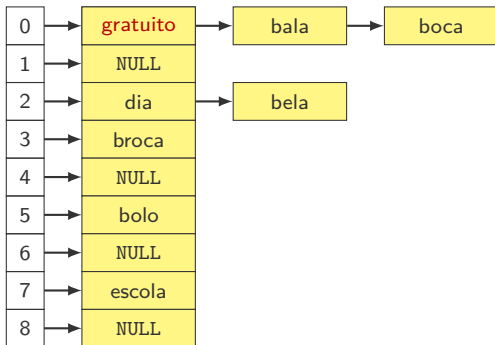
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola  $\rightsquigarrow h(\text{"escola"}) = 7$

gratuito  $\rightsquigarrow h(\text{"gratuito"}) = 0$

ilha



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

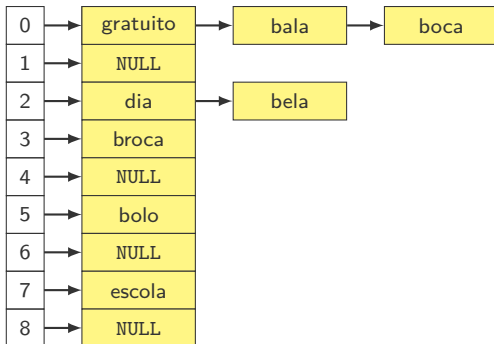
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola  $\rightsquigarrow h(\text{"escola"}) = 7$

gratuito  $\rightsquigarrow h(\text{"gratuito"}) = 0$

ilha  $\rightsquigarrow h(\text{"ilha"}) = 6$



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

A classe **HashSet** é implementada usando uma **tabela hash**

broca  $\rightsquigarrow h(\text{"broca"}) = 3$

boca  $\rightsquigarrow h(\text{"boca"}) = 0$

bolo  $\rightsquigarrow h(\text{"bolo"}) = 5$

bela  $\rightsquigarrow h(\text{"bela"}) = 2$

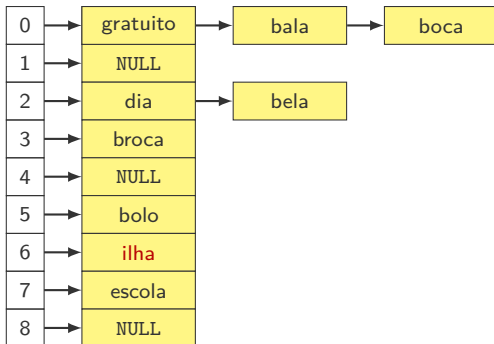
bala  $\rightsquigarrow h(\text{"bala"}) = 0$

dia  $\rightsquigarrow h(\text{"dia"}) = 2$

escola  $\rightsquigarrow h(\text{"escola"}) = 7$

gratuito  $\rightsquigarrow h(\text{"gratuito"}) = 0$

ilha  $\rightsquigarrow h(\text{"ilha"}) = 6$



- Ideia:** Dada uma tabela com  $M = 9$  slots, usamos uma **função hash** para associar cada chave a um número inteiro (entre 0 e 8)

# Tabela hash

- Cada slot do array é chamado de **balde** (*bucket*).
- O número de slots  $M$  é chamado **capacidade** (*capacity*).
- O **fator de carga** (*load factor*) de uma tabela hash é o valor  $\alpha = n/M$ , onde  $n$  é o número de chaves armazenadas e  $M$  é a capacidade.
  - Quando o fator de carga se aproxima do número 0.75, o Java cria uma tabela hash com uma capacidade maior e insere os elementos da tabela antiga na nova tabela. Essa operação é chamada **rehashing**.



# Classe HashSet

- A classe `HashSet` possui todos os métodos definidos na superclasse `Object` e implementa todos os métodos declarados nas interfaces `Collection`, `Iterable` e `Set`.
- Essa classe possui 4 construtores:
  - `HashSet()`: constrói um conjunto vazio. A tabela hash interna possui capacidade inicial de 16 e fator de carga 0.75
  - `HashSet(int initialCapacity)`: constrói um novo conjunto vazio. A tabela hash interna possui a capacidade especificada e fator de carga 0.75
  - `HashSet(int initialCapacity, float loadFactor)`: constrói um novo conjunto vazio. A tabela hash interna possui a capacidade e o fator de carga especificados.
  - `HashSet(Collection<E> c)`: constrói um novo conjunto contendo os elementos da coleção especificada.

# Exemplos

- Analisar o projeto [OperacoesEmConjuntos](#).

# Classe TreeSet



# Classe TreeSet

- A classe `TreeSet` é similar à classe `HashSet`, porém ela possui algumas diferenças.
- `TreeSet` implementa diretamente a interface `NavigableSet`. Logo, ela implementa indiretamente as interfaces `SortedSet` e `Set`.

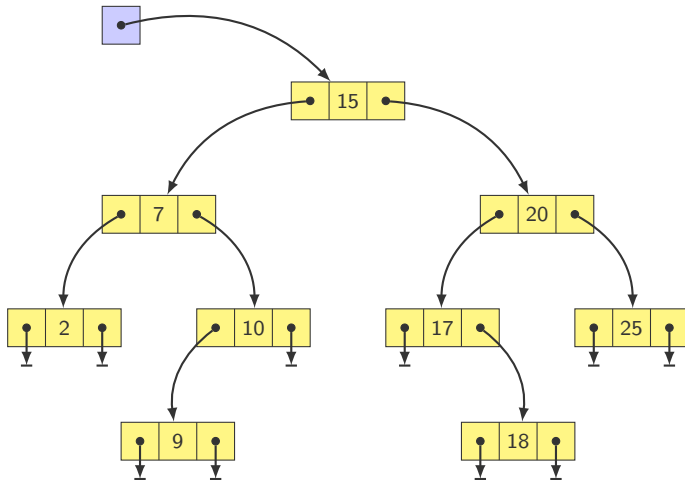
# Classe TreeSet

- A classe **TreeSet** é similar à classe **HashSet**, porém ela possui algumas diferenças.
- **TreeSet** implementa diretamente a interface **NavigableSet**. Logo, ela implementa indiretamente as interfaces **SortedSet** e **Set**.
- Por implementar **SortedSet**, seus elementos são ordenados independente da ordem que você inserir os elementos.
  - Isso significa que se você usar um iterador para navegar pelos elementos do conjunto, eles aparecerão em ordem.
  - Porém, isso tem um custo. As operações **add**, **remove** e **contains** são mais lentas aqui do que em um **HashSet**.

# Classe TreeSet

- A classe **TreeSet** é similar à classe **HashSet**, porém ela possui algumas diferenças.
- **TreeSet** implementa diretamente a interface **NavigableSet**. Logo, ela implementa indiretamente as interfaces **SortedSet** e **Set**.
- Por implementar **SortedSet**, seus elementos são ordenados independente da ordem que você inserir os elementos.
  - Isso significa que se você usar um iterador para navegar pelos elementos do conjunto, eles aparecerão em ordem.
  - Porém, isso tem um custo. As operações **add**, **remove** e **contains** são mais lentas aqui do que em um **HashSet**.
- Além disso, ao invés de tabela hash, a classe **TreeSet** usa uma **árvore binária de busca** como estrutura de dado interna.

# Ilustração de uma árvore binária de busca



# Classe TreeSet<E> — Métodos

`TreeSet<E>` implementa todos os métodos da interface `Set`. Além disso, `TreeSet<E>` contém métodos adicionais, não presentes em `HashSet`, que operam no conjunto considerando a ordem dos elementos contidos nele.



# Classe TreeSet<E> — Métodos

`TreeSet<E>` implementa todos os métodos da interface `Set`. Além disso, `TreeSet<E>` contém métodos adicionais, não presentes em `HashSet`, que operam no conjunto considerando a ordem dos elementos contidos nele.

- `E first()`: retorna o primeiro (menor) elemento contido no conjunto como uma instância do mesmo tipo dos elementos contidos na lista.
- `E last()`: retorna o último (maior) elemento contido no conjunto como uma instância do mesmo tipo dos elementos contidos na lista.

# Classe TreeSet<E>

A classe `TreeSet<E>` implementa todos os métodos da interface `Set`. Além disso, ela possui métodos adicionais, alguns dos quais estão abaixo:

- `E ceiling(E e)`: retorna o menor elemento do conjunto que é maior que ou igual ao elemento dado, ou `null` caso não exista tal elemento.

# Classe TreeSet<E>

A classe `TreeSet<E>` implementa todos os métodos da interface `Set`. Além disso, ela possui métodos adicionais, alguns dos quais estão abaixo:

- `E ceiling(E e)`: retorna o menor elemento do conjunto que é maior que ou igual ao elemento dado, ou `null` caso não exista tal elemento.
- `E floor(E e)`: retorna o maior elemento do conjunto que é menor que ou igual ao elemento dado, ou `null` caso não exista tal elemento.

# Classe TreeSet<E>

A classe `TreeSet<E>` implementa todos os métodos da interface `Set`. Além disso, ela possui métodos adicionais, alguns dos quais estão abaixo:

- `E ceiling(E e)`: retorna o menor elemento do conjunto que é maior que ou igual ao elemento dado, ou `null` caso não exista tal elemento.
- `E floor(E e)`: retorna o maior elemento do conjunto que é menor que ou igual ao elemento dado, ou `null` caso não exista tal elemento.
- `E pollFirst()`: retorna e remove o menor elemento, ou retorna `null` se o conjunto estiver vazio.

# Classe TreeSet<E>

A classe `TreeSet<E>` implementa todos os métodos da interface `Set`. Além disso, ela possui métodos adicionais, alguns dos quais estão abaixo:

- `E ceiling(E e)`: retorna o menor elemento do conjunto que é maior que ou igual ao elemento dado, ou `null` caso não exista tal elemento.
- `E floor(E e)`: retorna o maior elemento do conjunto que é menor que ou igual ao elemento dado, ou `null` caso não exista tal elemento.
- `E pollFirst()`: retorna e remove o menor elemento, ou retorna `null` se o conjunto estiver vazio.
- `E pollLast()`: retorna e remove o maior elemento, ou retorna `null` se o conjunto estiver vazio.

# Classe TreeSet<E>

- `SortedSet<E> headSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente menores que `element`.

## Classe TreeSet<E>

- `SortedSet<E> headSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente menores que `element`.
- `SortedSet<E> tailSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente maiores que ou iguais a `element`.

## Classe TreeSet<E>

- `SortedSet<E> headSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente menores que `element`.
- `SortedSet<E> tailSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente maiores que ou iguais a `element`.
- `SortedSet<E> subSet(E fromElem, E toElem)`: retorna uma **visão** da porção deste conjunto cujos elementos vão de `fromElem`, inclusive, até `toElem`, exclusive.



## Classe TreeSet<E>

- `SortedSet<E> headSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente menores que `element`.
- `SortedSet<E> tailSet(E element)`: retorna uma **visão** da porção deste conjunto cujos elementos são estritamente maiores que ou iguais a `element`.
- `SortedSet<E> subSet(E fromElem, E toElem)`: retorna uma **visão** da porção deste conjunto cujos elementos vão de `fromElem`, inclusive, até `toElem`, exclusive.

Para mais detalhes, consulte a API do Java: <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/TreeSet.html>

# Exemplos

- Analisar o projeto [NavigableSet Test](#).
- Analisar o projeto [Loteria](#).

# Interface Map<K, V>

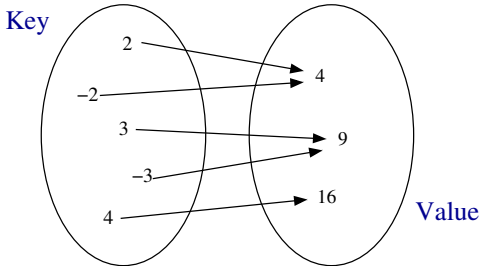


# Map

- **Map** é uma estrutura de dados que armazena pares (chave, valor)
  - chave e valor são objetos
  - Não permite chaves repetidas, mas admite valores repetidos
  - Os elementos são indexados pelo objeto chave (não possuem repetição)
  - Acesso, inserção e remoção de elementos são **rápidos**:  $O(1)$

Pares:

(2, 4)  
(-2, 4)  
(3, 9)  
(-3, 9)  
(4, 16)



# Interface Map<K, V>

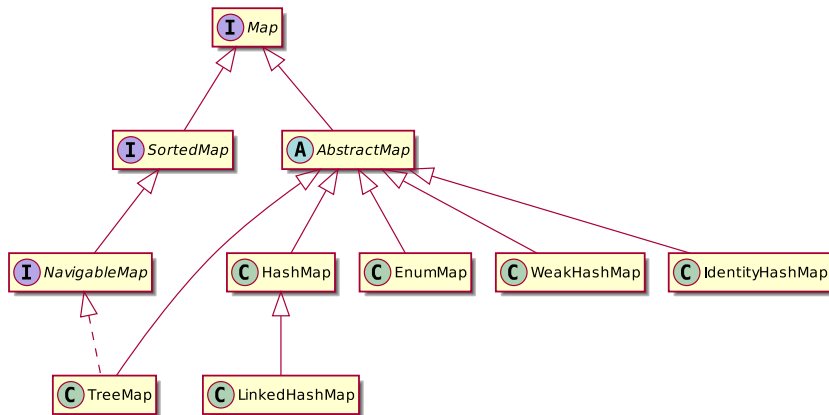
- Em Java, os métodos que podem ser aplicados a um map são definidos na interface `Map<K, V>` do pacote `java.util`.

# Interface Map<K, V>

- Em Java, os métodos que podem ser aplicados a um map são definidos na interface `Map<K, V>` do pacote `java.util`.
- Map<K, V> possui duas implementados principais: `HashMap` e `TreeMap`.
  - `HashMap` usa uma tabela hash como estrutura interna, que garante melhor performance para operações básicas (acesso, inserção e remoção).
  - `TreeMap` usa uma árvore binária de busca como estrutura interna, que garante que as chaves dos objetos estarão em sua ordem natural, à custa de alguma performance.

# Interface Map<K, V>

- Em Java, os métodos que podem ser aplicados a um map são definidos na interface `Map<K,V>` do pacote `java.util`.
- Map<K,V> possui duas implementados principais: `HashMap` e `TreeMap`.
  - `HashMap` usa uma tabela hash como estrutura interna, que garante melhor performance para operações básicas (acesso, inserção e remoção).
  - `TreeMap` usa uma árvore binária de busca como estrutura interna, que garante que as chaves dos objetos estarão em sua ordem natural, à custa de alguma performance.
- Ambas as classes têm construtores vazios e construtores que recebem uma instância de qualquer classe que implemente a interface Map, usando os pares de chaves e valores nestes mapas passados como argumentos para inicializar as instâncias.



Hierarquia de classes — Map



## Interface Map<K, V> — Métodos

- `V put(K key, V value)`: Associa o valor especificado com a chave especificada e salva no map. Se o map já possuía um valor associado a esta chave, o valor antigo é substituído pelo novo.

## Interface Map<K, V> — Métodos

- `V put(K key, V value)`: Associa o valor especificado com a chave especificada e salva no map. Se o map já possuía um valor associado a esta chave, o valor antigo é substituído pelo novo.
- `void putAll(Map<K, V> m)`: copia todos os mapeamentos de `m` para este map.

## Interface Map<K,V> — Métodos

- `V put(K key, V value)`: Associa o valor especificado com a chave especificada e salva no map. Se o map já possuía um valor associado a esta chave, o valor antigo é substituído pelo novo.
- `void putAll(Map<K,V> m)`: copia todos os mapeamentos de `m` para este map.
- `V get(Object key)`: retorna o valor para o qual a chave especificada foi mapeada, ou retorna `null` se este map não contém nenhum mapeamento para esta chave.

## Interface Map<K,V> — Métodos

- `V put(K key, V value)`: Associa o valor especificado com a chave especificada e salva no map. Se o map já possuía um valor associado a esta chave, o valor antigo é substituído pelo novo.
- `void putAll(Map<K,V> m)`: copia todos os mapeamentos de `m` para este map.
- `V get(Object key)`: retorna o valor para o qual a chave especificada foi mapeada, ou retorna `null` se este map não contém nenhum mapeamento para esta chave.
- `boolean containsKey(Object key)`: retorna `true` se este map contém um mapeamento para a chave especificada.

## Interface Map<K,V> — Métodos

- `V put(K key, V value)`: Associa o valor especificado com a chave especificada e salva no map. Se o map já possuía um valor associado a esta chave, o valor antigo é substituído pelo novo.
- `void putAll(Map<K,V> m)`: copia todos os mapeamentos de `m` para este map.
- `V get(Object key)`: retorna o valor para o qual a chave especificada foi mapeada, ou retorna `null` se este map não contém nenhum mapeamento para esta chave.
- `boolean containsKey(Object key)`: retorna `true` se este map contém um mapeamento para a chave especificada.
- `boolean containsValue(Object value)`: retorna `true` se este map mapeia uma ou mais chaves para o valor especificado.

## Interface Map<K,V> — Métodos

- `V put(K key, V value)`: Associa o valor especificado com a chave especificada e salva no map. Se o map já possuía um valor associado a esta chave, o valor antigo é substituído pelo novo.
- `void putAll(Map<K,V> m)`: copia todos os mapeamentos de `m` para este map.
- `V get(Object key)`: retorna o valor para o qual a chave especificada foi mapeada, ou retorna `null` se este map não contém nenhum mapeamento para esta chave.
- `boolean containsKey(Object key)`: retorna `true` se este map contém um mapeamento para a chave especificada.
- `boolean containsValue(Object value)`: retorna `true` se este map mapeia uma ou mais chaves para o valor especificado.

## Interface Map<K,V> — Métodos

- `V remove(Object key)`: remove o mapeamento para esta chave.

## Interface Map<K, V> — Métodos

- `V remove(Object key)`: remove o mapeamento para esta chave.
- `V replace(K key, V value)`: substitui a entrada para a chave especificada somente se ela estiver atualmente mapeada para algum valor.



## Interface Map<K, V> — Métodos

- `V remove(Object key)`: remove o mapeamento para esta chave.
- `V replace(K key, V value)`: substitui a entrada para a chave especificada somente se ela estiver atualmente mapeada para algum valor.
- `Collection<V> values()`: retorna uma **visão** dos valores contidos neste map, como uma `Collection<V>`.

## Interface Map<K, V> — Métodos

- `V remove(Object key)`: remove o mapeamento para esta chave.
- `V replace(K key, V value)`: substitui a entrada para a chave especificada somente se ela estiver atualmente mapeada para algum valor.
- `Collection<V> values()`: retorna uma **visão** dos valores contidos neste map, como uma `Collection<V>`.
- `Set<K> keySet()`: retorna uma **visão** das chaves contidas neste mapa, como um `Set<K>`.
- `int size()`: retorna a quantidade de pares chave-valor neste map.

## Interface Map<K, V> — Métodos

- `V remove(Object key)`: remove o mapeamento para esta chave.
- `V replace(K key, V value)`: substitui a entrada para a chave especificada somente se ela estiver atualmente mapeada para algum valor.
- `Collection<V> values()`: retorna uma **visão** dos valores contidos neste map, como uma `Collection<V>`.
- `Set<K> keySet()`: retorna uma **visão** das chaves contidas neste mapa, como um `Set<K>`.
- `int size()`: retorna a quantidade de pares chave-valor neste map.
- `void clear()`: remove todos os mapeamentos deste map.

# Interface Map<K, V> — Métodos

- `boolean isEmpty()`: retorna `true` se este map estiver vazio.
- Visite a API do Java para mais detalhes: <https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Map.html>

## Classe HashMap<K,V> — Construtores

- `HashMap()`: constrói uma tabela hash vazia com capacidade inicial 16 e fator de carga 0.75
- `HashMap(int initialCapacity)`: constrói uma tabela hash vazia com a capacidade inicial especificada e com fator de carga 0.75
- `HashMap(int initialCapacity, float loadFactor)`: constrói uma tabela hash vazia com a capacidade inicial e fator de carga especificados.
- `HashMap(Map<K,V> m)`: constrói uma nova tabela hash contendo os mesmos mapeamentos que `m`.

# Classe TreeMap<K,V> — Construtores

- `TreeMap()`: constrói uma TreeMap vazia.
- `TreeMap(Comparator<K> comparator)`: constrói uma nova TreeMap vazia, baseada na ordenação do comparador dado como argumento.
- `TreeMap(Map<K,V> m)`: constrói uma nova TreeMap contendo os mesmos mapeamentos que o Map `m`, ordenados de acordo com a ordenação natural das suas chaves.
- `TreeMap(SortedMap<K,V> m)`: constrói uma nova TreeMap contendo os mesmos mapeamentos e usando a mesma ordenação que o SortedMap `m`.

# Exemplos

- Mostrar projeto [Map\\_Project01](#)
- Mostrar projeto [Map\\_Project02](#)
- Mostrar projeto [Map\\_Project03](#)

FIM

