



# Usando Collection - Parte 1



## ollection


Uma estrutura de dados é uma coleção de dados organizados de alguma maneira. A estrutura não apenas armazena dados, mas também suporta operações para acessar e manipular os dados. O pacote `java.util` contém um dos subsistemas mais poderosos do Java: The Collections Framework. O Framework de coleções é uma hierarquia sofisticada de interfaces e classes que fornece tecnologia de ponta para gerenciar grupos de objetos.

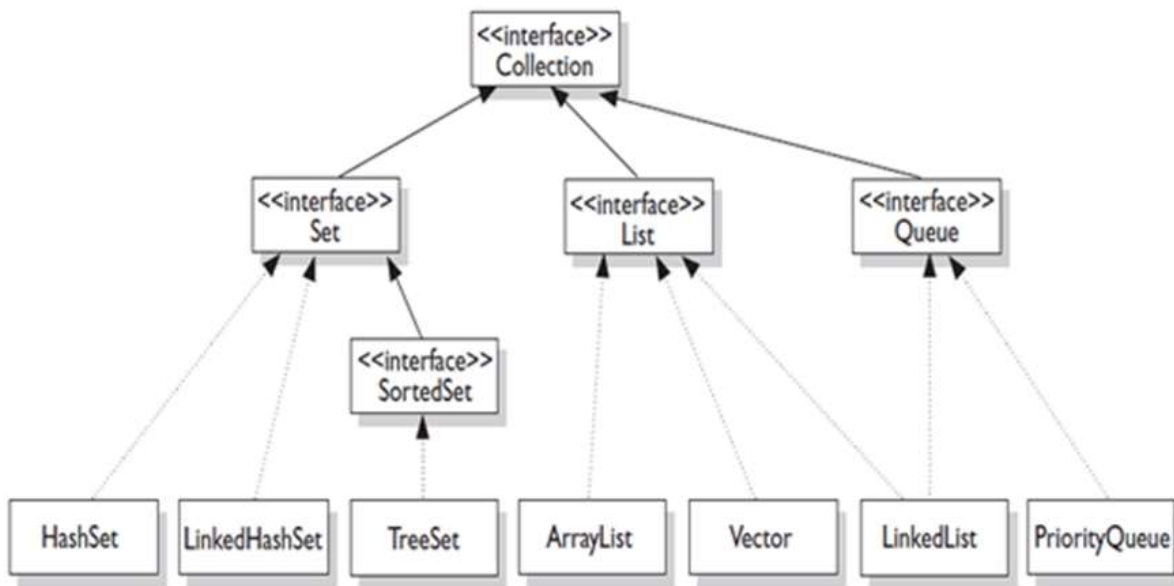
Você pode executar as seguintes atividades usando a estrutura de coleta Java,

- Adicionar objetos à coleção
- Remover objetos da coleção
- Procurar um objeto na coleção
- Recuperar / obter objeto da coleção
- Repita a coleção para obter funcionalidades específicas da empresa.

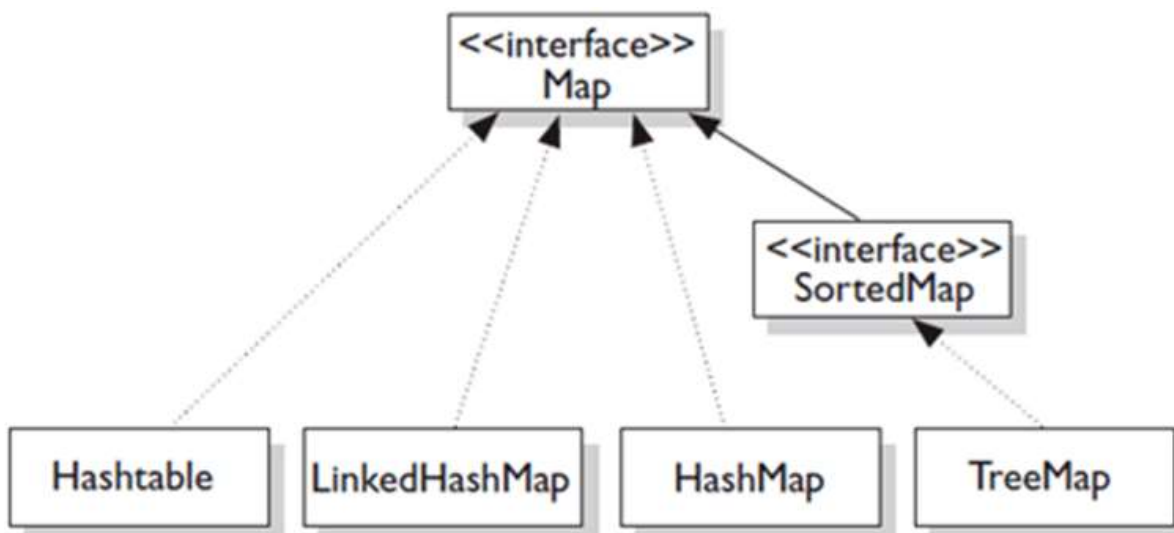
## Interfaces-chave e classes de estrutura de Collection

- **coleção** (c minúsculo): representa qualquer uma das estruturas de dados nas quais os objetos são armazenados e iterados.
- **Coleção** (capital C): na verdade, é a interface `java.util.Collection` da qual `Set`, `List` e `Queue` se estendem.

**Coleções** (C maiúsculo e termina com s): é a classe `java.util.Collections` que mantém uma pilha de métodos de utilidade estática para uso com coleções. 



Existem algumas outras classes na estrutura de coleção que não estendem a Interface de Coleção, elas implementam a interface de Mapa.



Podemos dizer que a coleção possui 4 tipos básicos:

- List:

A interface List estende a Coleção para definir uma coleção ordenada com duplicatas permitidas. A interface List adiciona operações orientadas à posição, bem como um novo iterador de lista que permite ao usuário percorrer a lista bidirecionalmente. ArrayList, LinkedList e Vector são classes que implementam a interface List.



- Set:

A interface Set estende a interface Collection. Isso garantirá que uma instância de Set não contenha elementos duplicados. A classe concreta implementa códigos de hash e métodos iguais para garantir a exclusividade dos objetos. Três classes concretas de Set são HashSet, LinkedHashSet e TreeSet.

- Map:


O Map é um contêiner que armazena os elementos junto com as chaves. As chaves são como índices. No List, os índices são números inteiros. No Map, as chaves podem ser qualquer objeto. O Map não pode conter chaves duplicadas. Cada chave é mapeada para um valor. Uma chave e seu valor correspondente de uma entrada, que é realmente armazenada em um mapa. HashMap, Hashtable, TreeMap e LinkedHashMap são classes que implementam a interface do Map.

- Queue:

Queue é uma estrutura de dados que entra e sai. Os elementos são anexados ao final da fila e são removidos do início da fila. Em uma fila de prioridades, os elementos recebem prioridades. Ao acessar elementos, o elemento com a maior prioridade é removido primeiro.

Podemos ter subtipos de classes de coleção como classificadas, não-ordenadas, ordenadas e não-ordenadas.

- Ordenado: quando uma coleção é solicitada, significa que você pode percorrer a coleção em uma ordem específica (não aleatória)

- **Classificação:** uma coleção classificada significa que a ordem dos objetos na coleção é determinada de acordo com alguma regra ou regras, conhecida como ordem de classificação. Uma ordem de classificação não tem nada a ver com quando um objeto foi adicionado à coleção, ou quando foi a última vez que foi acessado ou em qual “posição” foi adicionada. 

## Interface Iterator

O Iterator permite percorrer uma coleção, obtendo ou removendo elementos. ListIterator estende o Iterator para permitir a passagem bidirecional de uma lista e a modificação de elementos.

Um iterador é um objeto associado a uma coleção específica. Permite percorrer a coleção passo a passo. Existem dois métodos importantes do iterador.

- **boolean hasNext():** Retorna true se houver pelo menos mais um elemento na coleção que está sendo percorrida. A chamada de Next () NÃO o move para o próximo elemento da coleção.
- **object next():** este método retorna o próximo objeto na coleção E move-o para frente até o elemento após o retorno do elemento.

## Interface do Comparator

A interface Comparator oferece a capacidade de classificar uma determinada coleção de várias maneiras diferentes. A outra coisa útil sobre a interface do Comparator é que você pode usá-lo para classificar instâncias de qualquer classe - mesmo classes que não podem ser modificadas - ao contrário da interface Comparável, que o força a alterar a classe cujas instâncias você deseja classificar.

A interface Comparator também é muito fácil de implementar, tendo apenas um método, compare (). O método Comparator.compare () retorna um int.

## Assinatura do método

`int compare (objOne, objTwo)`



O método `Compare ()` retorna


- negativo se `objOne < objTwo`
- zero se `objOne == objTwo`
- positivo se `objOne > objTwo`

## Classe `ArrayList` e `Vector`

Além da classe `Arrays`, o Java fornece uma classe `ArrayList` que pode ser usada para criar contêineres que armazenam listas de objetos. `ArrayList` pode ser considerado como uma matriz expansível. Oferece iteração rápida e acesso aleatório rápido. `ArrayList` implementa a nova interface `RandomAccess` - uma interface de marcador (o que significa que não possui métodos) que diz: “Esta lista suporta acesso aleatório rápido (geralmente em tempo constante)”. Escolha isso em um `LinkedList` quando você precisar de iteração rápida, mas provavelmente não fará muita inserção e exclusão.

As versões anteriores do Java possuem uma classe de coleção herdada chamada `Vector`, que é muito semelhante à `ArrayList`. O vetor implementa uma matriz dinâmica. Um vetor é basicamente o mesmo que um `ArrayList`, mas os métodos de vetor são sincronizados para segurança do encadeamento. Você normalmente deseja usar `ArrayList` em vez de `Vector`, porque os métodos sincronizados adicionam uma ocorrência de desempenho que talvez você não precise.

A classe `java.util.ArrayList` é uma das classes mais comumente usadas no Framework Framework. A `ArrayList` é redimensionável dinamicamente, o que significa que seu tamanho pode mudar durante a execução do programa. Isso significa que:

- Você pode adicionar um item a qualquer momento em um contêiner ArrayList e o tamanho da matriz se expande automaticamente para acomodar o novo item 
- Você pode remover um item a qualquer momento em um contêiner ArrayList e o tamanho da matriz se contrai automaticamente.

Para afirmar o óbvio: ArrayList é uma coleção ordenada (por índice), mas não classificada.

Para usar a classe ArrayList, você deve usar a seguinte instrução de importação:

```
import java.util.ArrayList;
```

Em seguida, para declarar um ArrayList, você pode usar o construtor padrão, como no exemplo a seguir:

```
ArrayList nomes = new ArrayList();
```

O construtor padrão cria um ArrayList com capacidade para 10 itens. A capacidade de um ArrayList é o número de itens que ele pode armazenar sem precisar aumentar seu tamanho. Outros construtores de ArrayList da seguinte maneira,

```
ArrayList nomes = new ArrayList(int size);
```

```
ArrayList nomes = new ArrayList(Collection c);
```

Você também pode especificar a capacidade / tamanho do ArrayList inicial, bem como criar ArrayList a partir de outros tipos de coleção.

Algumas das vantagens que o ArrayList tem sobre os arrays são

- Pode crescer dinamicamente.



- Ele fornece mecanismos de inserção e pesquisa mais poderosos do que matrizes.

## Métodos ArrayList

Método	Objetivo
<code>public void add (Object)</code> <code>public void add (int, Object)</code>	Adicionar um item a um ArrayList. A versão padrão adiciona um item no próximo local disponível; uma versão sobrecarregada permite especificar uma posição na qual adicionar o item
<code>public void remove(int)</code>	Remove um item de um ArrayList em um local especificado
<code>public void set(int, Object)</code>	Alterar um item em um local especificado de ArrayList
<code>Object get(int)</code>	Recuperar um item de um local especificado em um ArrayList
<code>public int size()</code>	Retorna o tamanho atual de ArrayList

O seguinte programa demonstra o uso de todos os métodos descritos acima. Aqui estamos criando ArrayList chamado `minhaLista` e adicionando objetos usando o método `add ()`, bem como usando o método `add` baseado em índice, e imprimindo todos os objetos usando o loop `for`. Então, demonstramos o uso dos métodos `get ()`, `contains ()` e `size ()`. A saída do programa é mostrada abaixo do código Java.

```
import java.util.ArrayList;
```

```
public class ArrayListDemo {
```

```
public static void main(String[] args) {
```



```
    //declarando ArrayList de objetos String
```

```
    ArrayList<String> minhaLista = new ArrayList<String>();
```

```
    //Adicionando objetos à lista de matrizes no índice padrão
```

```
    minhaLista.add("Maçã");
```

```
    minhaLista.add("Manga");
```

```
    minhaLista.add("Laranja");
```

```
    minhaLista.add("Uvas");
```

```
    //Adicionando objeto a um índice específico
```

```
    minhaLista.add(1, "Laranja");
```

```
    minhaLista.add(2, "Abacaxi");
```

```
    System.out.println("Mostrar todos os objetos:");
```

```
    for(String s:minhaLista){
```

```
        System.out.println(s);
```

```
    }
```

```
        System.out.println("Adicionando objeto ao índice 3 "+  
minhaLista.get(3));
```

```
        System.out.println("Caqui está na lista: " +  
minhaLista.contains("Chicku"));
```

```
    System.out.println("Tamanho do ArrayList: " + minhaLista.size());
```



```
minhaLista.remove("Papaya");
```



```
System.out.println("Novo tamanho do ArrayList: "+
```

```
minhaLista.size());
```

```
}
```

```
}
```

## Resultado:

```
Mostrar todos os objetos:
```

```
Maçã
```

```
Laranja
```

```
Abacaxi
```

```
Manga
```

```
Laranja
```

```
Uvas
```

```
Adicionando objeto ao índice 3: Manga
```

```
Caqui está na lista: false
```

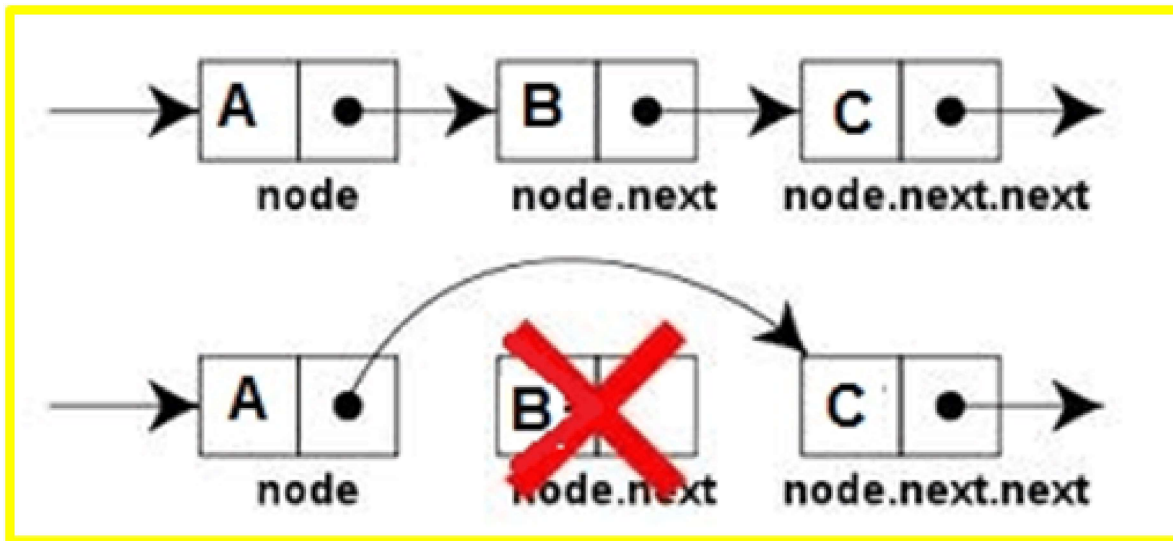
```
Tamanho do ArrayList: 6
```

```
Novo tamanho do ArrayList: 6
```

## Classe LinkedList

Um LinkedList é ordenado pela posição do índice, como ArrayList, exceto que os elementos estão duplamente vinculados um ao outro. Esse vínculo fornece novos métodos (além do que você obtém da interface de lista) para adicionar e remover do início ou do fim, o que facilita a implementação de uma pilha ou fila. A lista vinculada tem um conceito de nós e dados. Aqui o Nó está armazenando valores do próximo nó enquanto os dados armazenam o valor que está mantendo. O diagrama abaixo mostra como o LinkedList armazena valores. Existem três elementos no LinkedList A, B e C. Estamos removendo o elemento B do meio do

LinkedList que apenas vai alterar o valor do nó do elemento A para que aponte para o nó C.



Lembre-se de que um LinkedList pode iterar mais lentamente que um ArrayList, mas é uma boa opção quando você precisa de inserção e exclusão rápidas.

O LinkedList tem os dois construtores mostrados aqui:

```
iLinkedList()
```

```
LinkedList(Collection<? extends E> c)
```

O primeiro construtor cria uma lista vinculada vazia. O segundo construtor cria uma lista vinculada que é inicializada com os elementos da coleção c.

**Métodos importantes da classe LinkedList:**

Método	Descrição
addFirst () ou offerFirst ()	Para adicionar elementos ao início de uma lista
addLast () ou offerLast () ou add ()	Para adicionar elementos ao final da lista
getFirst () ou peekFirst ()	Para obter o primeiro elemento da lista
getLast () ou peekLast ()	Para obter o último elemento da lista
removeFirst () ou pollFirst () ou remove ()	Para remover o primeiro elemento da lista
removeLast () ou pollLast ()	Para remover o último elemento da lista

No seguinte programa demonstraremos o uso de todos os métodos descritos acima. Aqui, estamos criando o LinkedList chamado minhaLinkedList e adicionando objetos usando os métodos add (), addFirst () e addLast (), além de usar o método add () baseado em índice e imprimindo todos os objetos. Em seguida, modifique a lista usando os métodos remove (), removeLast () e remove (Object o). Em seguida, demonstramos o uso dos métodos getFirst () e getLast (). A saída do programa é mostrada abaixo do código Java.

```
import java.util.LinkedList;
```

```
public class LinkedListDemo {
```

```
    public static void main(String[] args) {
```

```
        LinkedList<String> minhaLinkedList = new LinkedList<String>();
```

```
        minhaLinkedList.addFirst("A");
```

```
        minhaLinkedList.add("B");
```

```
        minhaLinkedList.add("C");
```

```
minhaLinkedList.add("D");
```



```
minhaLinkedList.add(2, "X");//adiciona C ao index 2
```

```
minhaLinkedList.addLast("Z");
```

```
System.out.println("Lista antes de eliminar elementos");
```

```
System.out.println(minhaLinkedList);
```

```
minhaLinkedList.remove();
```

```
minhaLinkedList.removeLast();
```

```
minhaLinkedList.remove("C");
```

```
System.out.println("Lista depois de eliminar o primeiro e o último  
objeto");
```

```
System.out.println(minhaLinkedList);
```

```
System.out.println("Primeiro objeto: "+ minhaLinkedList.getFirst());
```

```
System.out.println("Último objeto: "+ minhaLinkedList.peekLast());
```

```
}
```

```
}
```

**Resultado:**

```
Lista antes de eliminar elementos
```

```
[A, B, X, C, D, Z]
```



```
Lista depois de eliminar o primeiro e o último  
objeto
```

```
[B, X, D]
```

```
Primeiro objeto: B
```


```
Último objeto: D
```

## HashSet

Um conjunto é uma coleção que não pode conter elementos duplicados. Ele modela a abstração do conjunto matemático. O HashSet estende o AbstractSet e implementa a interface Set. Ele cria uma coleção que usa uma tabela de hash para armazenamento. Uma tabela de hash armazena informações usando um mecanismo chamado hash. No hash, o conteúdo informativo de uma chave é usado para determinar um valor exclusivo, chamado de código de hash. O código hash é então usado como o índice no qual os dados associados à chave são armazenados. A transformação da chave em seu código de hash é realizada automaticamente.

Quando você coloca um objeto em um Hashset, ele usa o valor do código de hash do objeto para determinar onde colocar o objeto no conjunto. Mas também compara o código de hash do objeto com o código de hash de todos os outros objetos no conjunto de hash e, se não houver código de hash correspondente, o HashSet assume que esse novo objeto não é uma duplicata. O HashSet localiza um código de hash correspondente para dois objetos. um que você está inserindo e outro que já está no conjunto - o HashSet chamará um dos métodos equals () do objeto para verificar se esses objetos correspondentes a hashCode são realmente iguais. E se forem iguais, o HashSet sabe que o objeto que você está tentando adicionar é uma duplicata de algo no Conjunto, para que o add não aconteça.

Um HashSet é um conjunto não classificado e não ordenado. Ele usa o código de hash do objeto que está sendo inserido, portanto, quanto mais eficiente sua

implementação de hashCode (), melhor o desempenho de acesso que você terá. Use essa classe quando desejar uma coleção sem duplicatas e não se importo com a ordem, ao iterá-lo. 

HashSet () // Construtor padrão

HashSet (Collection c) // Cria o HashSet a partir da coleção c


HashSet (int capacidade) // cria o HashSet com a capacidade inicial mencionada

HashSet (int capacidade, float carregamento) // Cria o HashSet com capacidade e fator de carga

Escolher uma capacidade inicial muito alta pode desperdiçar espaço e tempo. Por outro lado, escolher uma capacidade inicial muito baixa desperdiça tempo ao copiar a estrutura de dados cada vez que é forçado a aumentar sua capacidade. Se você não especificar uma capacidade inicial, o padrão é 16.

A classe HashSet possui outro parâmetro de ajuste chamado fator de carga. O tamanho do HashSet aumenta conforme o fator de carga definido ou o tamanho duplo padrão.

## **Métodos HashSet**

Método	Objetivo
public boolean add(Object o)	Adicionar um objeto a um HashSet, se ,  não estiver presente no HashSet.
public boolean remove(Object o)	Remover um objeto de um HashSet se encontrado no HashSet.
public boolean contains(Object o)	Retorna true se o objeto encontrado retornar return false
public boolean isEmpty()	Retorna true se HashSet estiver vazio; caso contrário, retorna false
public int size()	Retorna o número de elementos no HashSet

```
import java.util.HashSet;
```

```
public class HashSetDemo {
```

```
public static void main(String[] args) {
```

```
    HashSet<String> hs = new HashSet<String>();
```

```
    // Adicionando elementos ao HashSet
```

```
    hs.add("M");
```

```
        hs.add("B");
```

```
        hs.add("C");
```

```
        hs.add("A");
```

```
        hs.add("M");
```

```
        hs.add("X");
```

```
        System.out.println("Tamanho do HashSet=" + hs.size());
```

```
System.out.println("HashSet Original:" + hs);
```



```
System.out.println("Remover A do HashSet: " +
```

```
hs.remove("A"));
```

```
System.out.println("Tentando remover Z, o qual não está  
presente: "
```

```
+ hs.remove("Z"));
```

```
System.out.println("Verificando se M está presente=" +
```

```
hs.contains("M"));
```

```
System.out.println("HashSet atualizado: " + hs);
```

```
}
```

```
}
```

### Resultado:

```
Tamanho do HashSet=5  
HashSet Original:[A, B, C, M, X]  
Remover A do HashSet:true  
Tentando remover Z, o qual não está  
presente: false  
Verificando se M está presente=true  
HashSet atualizado:[B, C, M, X]
```

### TreeSet

O TreeSet é uma das duas coleções classificadas (a outra é o TreeMap). O TreeSet estende o AbstractSet e implementa a interface NavigableSet. Ele cria uma coleção



que usa uma árvore para armazenamento. Os objetos são armazenados em ordem crescente e ordenada, de acordo com a ordem natural. Opcionalmente, você pode construir um TreeSet com um construtor que permita dar à coleção suas próprias regras para o que o pedido deve ser (em vez de depender da ordem definida pela classe dos elementos) usando um Comparable ou Comparator.

Os tempos de acesso e recuperação são bastante rápidos, o que torna o TreeSet uma excelente opção ao armazenar grandes quantidades de informações classificadas que devem ser encontradas rapidamente. O TreeSet pode não ser usado quando nosso aplicativo requer modificação do conjunto em termos de adição frequente de elementos.

## Construtores


```
TreeSet (); // Construtor padrão
```

```
TreeSet(Collection<? extends E> c); // TreeSet da coleção C
```

```
TreeSet(Comparator<? super E> comp); // TreeSet com pedidos personalizados  
conforme o comparador
```

```
TreeSet (SortedSet <E> ss); // TreeSet que contém os elementos de ss.
```

## Métodos importantes da classe TreeSet

Método	Descrição
void add(Object o)	Adiciona o elemento especificado a este conjunto, se ele ainda não estiver presente. 
void clear()	Remove todos os elementos deste conjunto.
Object first()	Retorna o primeiro elemento (mais baixo) atualmente neste conjunto classificado.
Object last()	Retorna o último elemento (mais alto) atualmente neste conjunto classificado.
boolean isEmpty()	Retorna true se este conjunto não contém elementos.
boolean remove(Object o)	Remove o elemento especificado deste conjunto, se estiver presente.
SortedSet subSet(Object fromElement, Object toElement)	Retorna uma visualização da parte deste conjunto cujos elementos variam fromElement, inclusive, a toElement, exclusivo.

```
import java.util.TreeSet;
```

```
public class TreeSetDemo {
```

```
    public static void main(String[] args) {
```

```
        TreeSet<String> playerSet = new TreeSet<String>();
```

```
        playerSet.add("Sonia");
```

```
        playerSet.add("Zelia");
```

```
        playerSet.add("Marcos");
```

```
        playerSet.add("Belém");
```

```
        playerSet.add("Verônica");
```

```
playerSet.add("Juliana");
```



```
playerSet.add("Ismael");
```

```
playerSet.add("Uriel");
```

```
playerSet.add("Pedro");
```

```
playerSet.add("Vanessa");
```

```
    playerSet.add("Sonia"); // Esse é um elemento duplicado, logo, não será  
adicionado novamente
```

```
//abaixo imprimirá a lista em ordem alfabética
```

```
System.out.println("Set Original:" + playerSet);
```

```
System.out.println("Primeiro Nome: " + playerSet.first());
```

```
System.out.println("Último Nome: " + playerSet.last());
```

```
    TreeSet<String> newPlySet = (TreeSet<String>) playerSet.subSet("Marcos",  
"Vanessa");
```

```
System.out.println("Sub set " + newPlySet);
```

```
}
```

```
}
```

**Resultado:**

```
Set Original: [Belém, Ismael, Juliana, Marcos, Pedro, Sonia, Uriel, Vanessa, Verônica, Zelia]
Primeiro nome: Belém
Último nome: Zelia
Sub set: [Marcos, Pedro, Sonia, Uriel]
```



No exemplo acima, estamos criando o objeto TreeSet of String. A classe String está tendo uma interface comparável implementada pela biblioteca Java. Vamos pensar em um caso em que precisamos ter nossos próprios objetos para serem armazenados em Set e ordenar objetos de acordo com nossa regra. O exemplo abaixo mostra Cricketers como um objeto com duas propriedades name e battingPosicao. Queremos armazenar todos os objetos Cricketers de acordo com a posição definida de batedura, quando iteramos através da coleção e obtemos nomes de acordo com a posição de batedura.

### **Código Java (Cricketer.java)**

```
package br.com.java.aula;

public class Cricketer {

    private String nome;

    private int battingPosicao;

    Cricketer(String cricketerNome, int cBattingPosicao){

        this.nome = cricketerName;

        this.battingPosicao = cBattingPosicao;

    }
}
```

```
public String getName() {
```

```
    return nome;
```

```
}
```

```
public int getBattingPosicao() {
```

```
    return battingPosicao;
```

```
}
```

```
}
```



### **Código Java (CompareCricketer.java)**

Aqui estamos definindo regras de como organizar objetos Cricketer no TreeMap.

```
package br.com.java.aula;
```

```
import java.util.Comparator;
```

```
public class CompareCricketer implements Comparator <Cricketer> {
```

```
    @Override
```

```
    public int compare(Cricketer arg0, Cricketer arg1) {
```

```
        if(arg0.getBattingPosicao() > arg1.getBattingPosicao())
```

```
            return 1;
```

```
        else if (arg0.getBattingPosicao() < arg1.getBattingPosicao())
```

```
            return -1;
```

```
else return 0;
```



```
}
```

```
}
```

```
package br.com.java.aula;
```

```
import java.util.Iterator;
```

```
import java.util.TreeSet;
```

```
public class CustomTreeSetDemo {
```

```
    public static void main(String[] args) {
```

```
        TreeSet<Cricketer> playerSet = new TreeSet<Cricketer>(  
            new CompareCricketer());
```

```
        playerSet.add(new Cricketer("Sonia", 1));
```

```
        playerSet.add(new Cricketer("Zelia", 9));
```

```
        playerSet.add(new Cricketer("Marcos", 7));
```

```
        playerSet.add(new Cricketer("Belém", 8));
```

```
        playerSet.add(new Cricketer("Verônica", 2));
```

```
        playerSet.add(new Cricketer("Juliana", 4));
```

```
        playerSet.add(new Cricketer("Ismael", 10));
```

```
        playerSet.add(new Cricketer("Uriel", 11));
```

```
        playerSet.add(new Cricketer("Pedro", 5));
```

```
playerSet.add(new Cricketer("Vanessa", 3));
```



```
playerSet.add(new Cricketer("Raina", 6));
```

```
Iterator<Cricketer> it = playerSet.iterator();
```

```
while (it.hasNext()) {
```

```
    System.out.println(it.next().getName());
```

```
}
```


```
}
```

```
}
```

## Resultado:

```
Sonia  
Verônica  
Vanessa  
Juliana  
Pedro  
Raina  
Marcos  
Belém  
Zelia  
Ismael  
Uriel
```

## LinkedHashSet

Um `LinkedHashSet` é uma versão ordenada do `HashSet` que mantém uma lista duplamente vinculada em todos os elementos. Use esta classe em vez do `Hash`  quando se importar com a ordem da iteração. Quando você itera através de um `HashSet`, a ordem é imprevisível, enquanto um `LinkedHashSet` permite iterar pelos elementos na ordem em que foram inseridos. Ao percorrer o `LinkedHashSet` usando um iterador, os elementos serão retornados na ordem em que foram inseridos.

**Construtores `LinkedHashSet`**

`LinkedHashSet ()` // Construtor padrão

`LinkedHashSet (Collection c)` // Cria `LinkedHashSet` a partir da coleção `c`

`LinkedHashSet (int capacidade)` // cria `LinkedHashSet` com a capacidade inicial mencionada

`LinkedHashSet (int capacidade, float carregaFator)` // Isso cria o `LinkedHashSet` com capacidade e fator de carga

**Métodos `LinkedHashSet`**

Método	Objetivo
<code>public boolean add(Object o)</code>	Adicionar um objeto a um <code>LinkedHashSet</code> , se já não estiver presente no <code>HashSet</code> .
<code>public boolean remove(Object o)</code>	Remove um objeto do <code>LinkedHashSet</code> se encontrado no <code>HashSet</code> .
<code>public boolean contains(Object o)</code>	Retorna <code>true</code> se o objeto encontrado retornar <code>return false</code> .
<code>public boolean isEmpty()</code>	Retorna <code>true</code> se <code>LinkedHashSet</code> estiver vazio, caso contrário, retorna <code>false</code> .
<code>public int size()</code>	Retorna o número de elementos no <code>LinkedHashSet</code> .



```
import java.util.LinkedHashSet;
```



```
public class LinkedHashSetDemo {
```

```
    public static void main(String[] args) {
```

```
        LinkedHashSet<String> linkedset = new LinkedHashSet<String>();
```

```
        // Adicionando elementos ao LinkedHashSet
```

```
        linkedset.add("Maruti");
```

```
        linkedset.add("BMW");
```

```
        linkedset.add("Honda");
```

```
        linkedset.add("Audi");
```

```
        linkedset.add("Maruti"); //Esse não adicionará o novo elemento  
        pois Maruti já existe.
```

```
        linkedset.add("WalksWagon");
```

```
        System.out.println("Tamanho do LinkedHashSet=" +  
        linkedset.size());
```

```
        System.out.println("LinkedHashSet Original:" + linkedset);
```

```
        System.out.println("Removendo Audi do LinkedHashSet: " +  
        linkedset.remove("Audi"));
```

```
        System.out.println("Tentando remover Z, o qual não está presente:
```

```
"
```

```
        + linkedset.remove("Z"));
```

```
System.out.println("Verificando se Maruti está presente=" +  
linkedset.contains("Maruti"));
```



```
System.out.println("LinkedHashSet atualizada: " + linkedset);  
  
}  
  
}
```

### Resultado:

```
Tamanho do LinkedHashSet=5  
LinkedHashSet Original:[Maruti, BMW, Honda, Audi,  
WalkWagon]  
Removendo Audi do LinkedHashSet: true  
Tentando remover Z, o qual não está presente:  
false  
Verificando se Maruti está presente=true  
LinkedHashSet atualizada: [Maruti, BMW, Honda,  
WalkWagon]
```

Podemos usar o objeto Iterator para iterar através de nossa coleção. Enquanto iteramos, podemos adicionar ou remover objetos da coleção. O programa abaixo demonstra o uso do iterador na coleção LinkedHashSet.

```
package br.com.java.aula;  
  
import java.util.Iterator;  
  
import java.util.LinkedHashSet;  
  
import java.util.Set;
```

```
public class LinkedHashSetIterator {
```



```
    public static void main(String[] args) {
```

```
        Set<String> meuCricketerSet = new LinkedHashSet<String>();
```

```
        meuCricketerSet.add("Ariel");
```

```
        meuCricketerSet.add("Daniela");
```

```
        meuCricketerSet.add("Jussara");
```

```
        meuCricketerSet.add("Benjamin");
```

```
        meuCricketerSet.add("Flavio");
```

```
        meuCricketerSet.add("Marcio");
```

```
        meuCricketerSet.add("Valeria");
```

```
        meuCricketerSet.add("Ramon");
```

```
        Iterator<String> setIterator = meuCricketerSet.iterator();
```

```
        while(setIterator.hasNext()){
```

```
            System.out.println(setIterator.next());
```

```
        }
```

```
    }
```

**Resultado:**

Ariel  
Daniela  
Jussara  
Benjamim  
Flavio  
Marcio  
Valeria  
Ramon



### Atividade extra

Vídeo: “Dominando Collection”

Link: <https://www.youtube.com/watch?v=PeFDQtLBlt0>

### Referências Bibliográficas

BARNES, D. J. KOLLING, M. **Programação orientada a objetos com java: uma introdução prática usando o bluej**. 4.ed. Pearson: 2009.

FELIX, R. (Org.). **Programação orientada a objetos**. Pearson: 2017.

MEDEIROS, L. F. de. **Banco de dados: princípios e prática**. Intersaberes: 2013;

ORACLE. Java Documentation, 2021. **Documentação oficial da plataforma Java**. Disponível em: < <https://docs.oracle.com/en/java/> >.

**Ir para exercício**

