

Computação Orientada a Objetos

Profa. Patrícia R. Oliveira

Parte 9 – Coleções



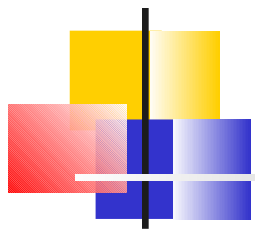
Como criar e manipular estruturas de dados?

- Abordagem mais “baixo nível”
 - Criar cada elemento de cada estrutura de dados e modificar essas estruturas manipulando diretamente seus elementos e as referências a seus elementos



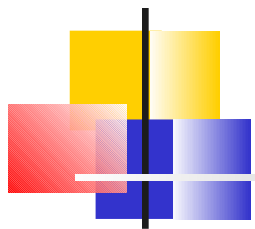
Como criar e manipular estruturas de dados?

- Abordagem mais “alto nível”
 - Utilizar a estrutura de coleções de Java, que contém estruturas de dados pré-empacotadas, interfaces e algoritmos para manipular essas estruturas



Coleções

- Com as coleções, os programadores utilizam estruturas de dados existentes, sem se preocupar com a maneira como elas estão implementadas
- É um bom exemplo de reutilização de código



Coleções

- A estrutura de coleções (**Collections Framework**) Java fornece componentes reutilizáveis prontos para utilização
- As coleções são padronizadas de modo que aplicativos possam compartilhá-las facilmente, sem preocupação com detalhes de implementação



Visão geral

- O que é uma coleção?
 - É uma estrutura de dados (um objeto) que agrupa referências a vários outros objetos
 - algumas vezes chamada de container
- Usadas para armazenar, recuperar e manipular elementos que formam um grupo natural (objetos do mesmo tipo)



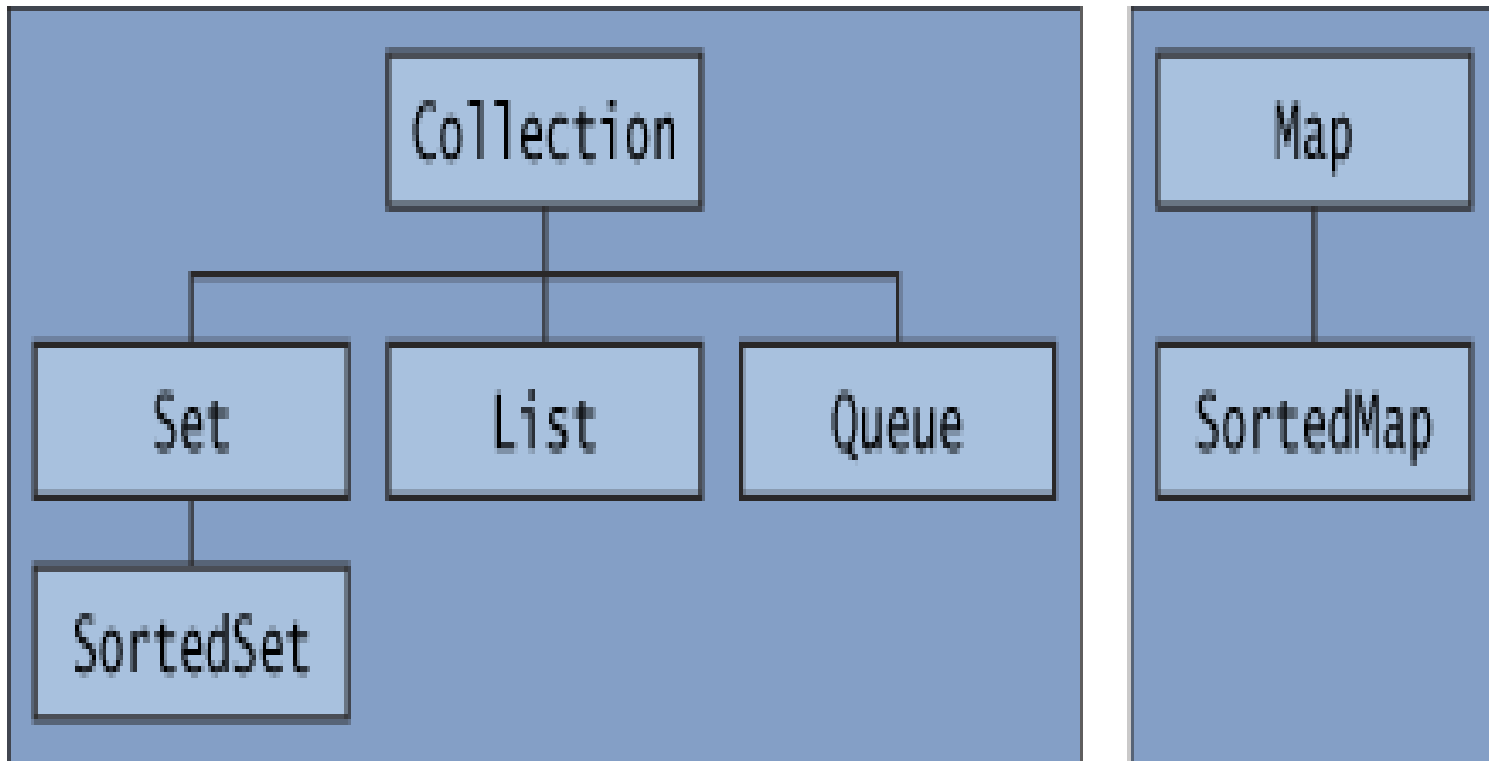
Visão geral

- As interfaces da estrutura de coleções (**Collections Framework**) Java declaram operações a serem realizadas genericamente em vários tipos de coleções



Interfaces da estrutura de coleções

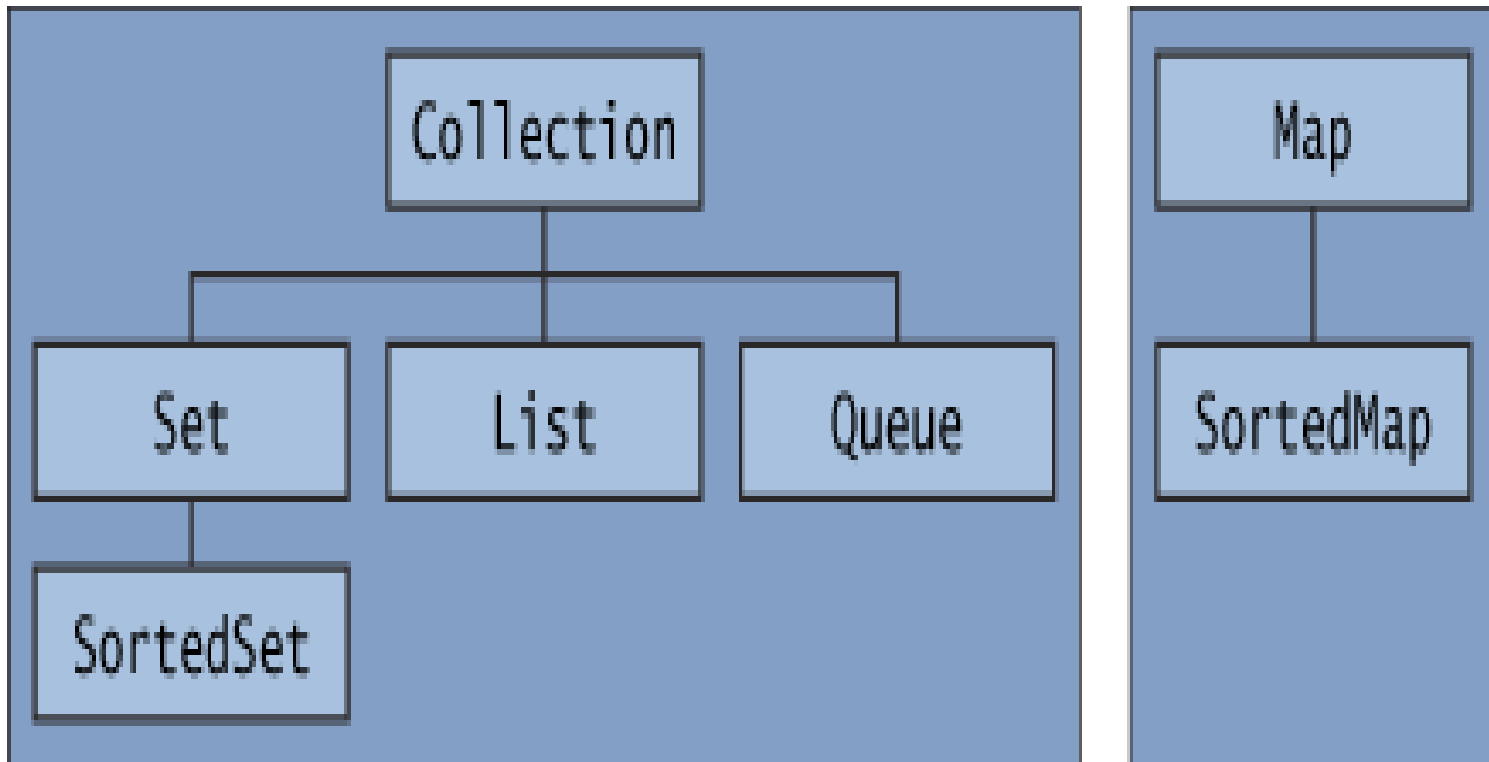
Interface **Collection**: raiz da hierarquia de coleções





Interfaces da estrutura de coleções

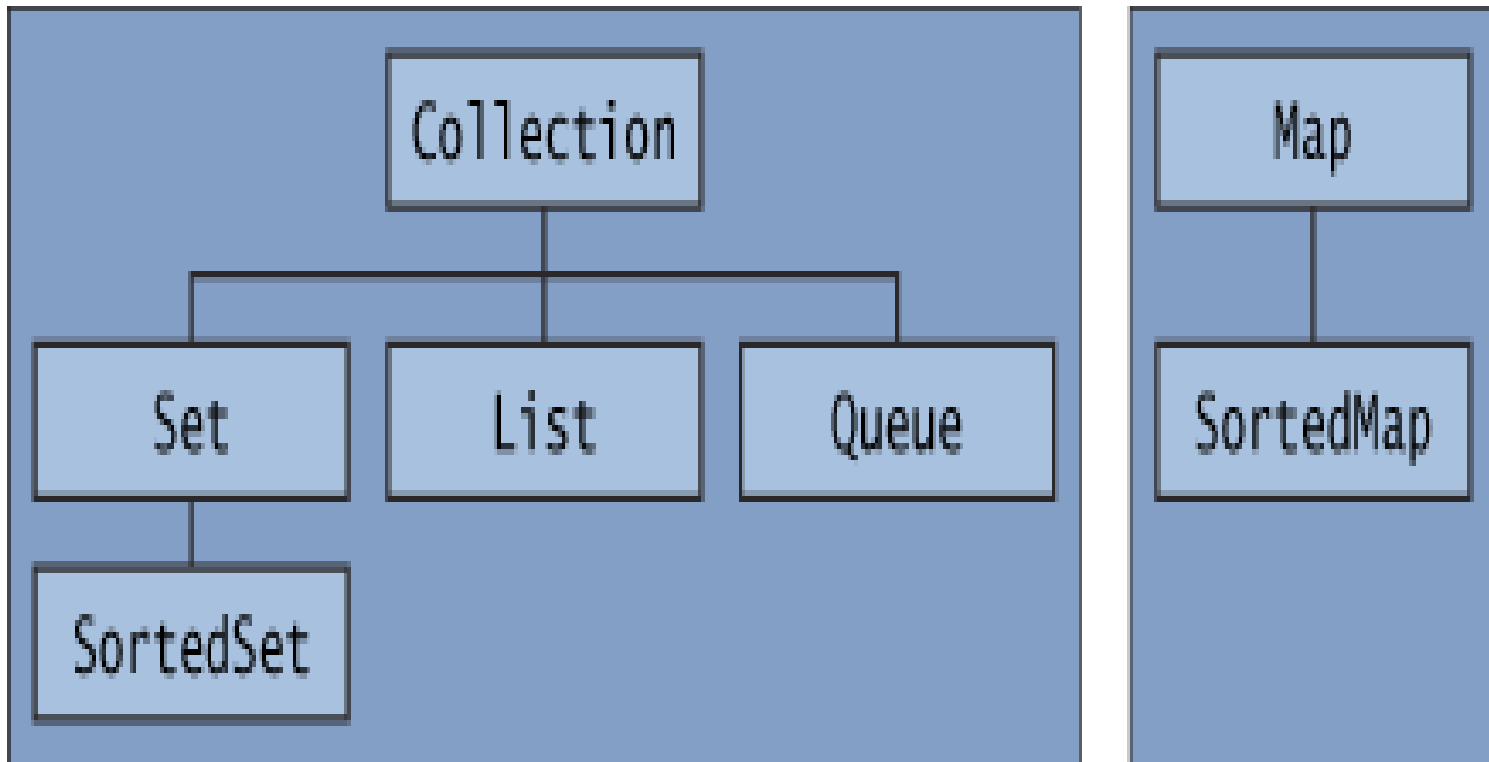
Interface **Set**: coleção que não contém duplicatas





Interfaces da estrutura de coleções

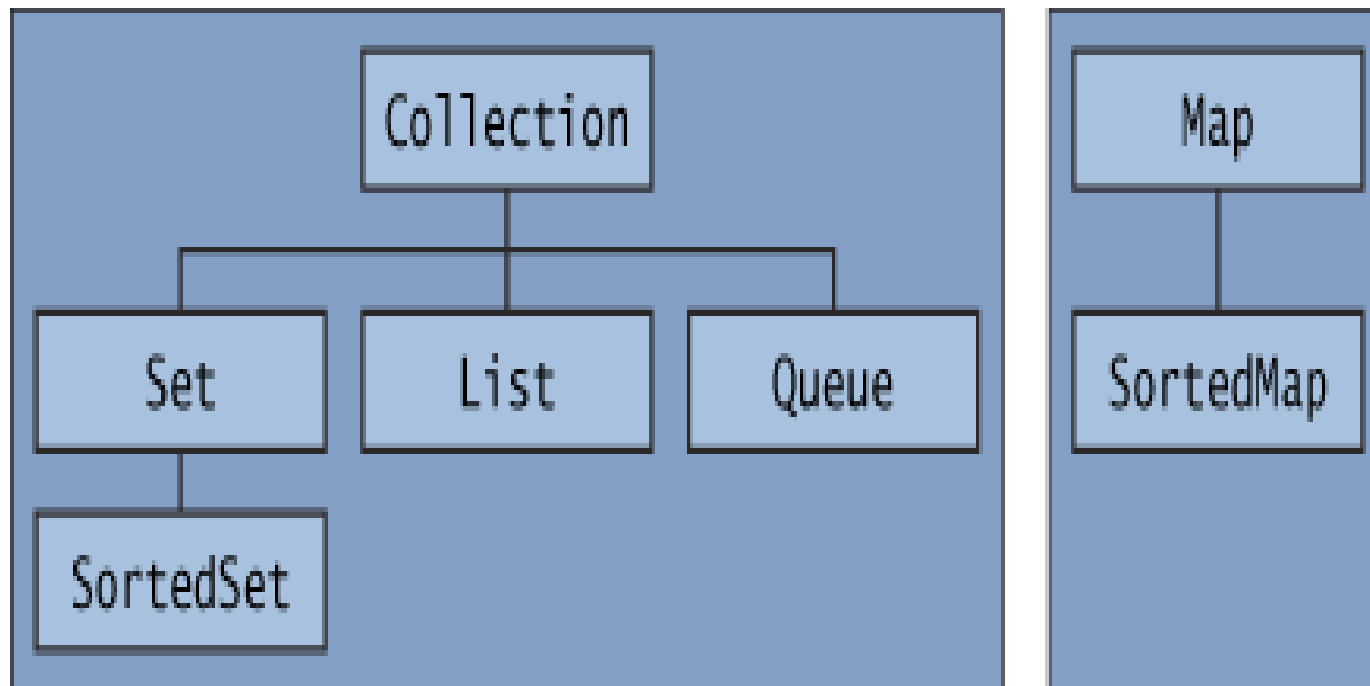
Interface **List**: coleção ordenada que pode conter elementos duplicados





Interfaces da estrutura de coleções

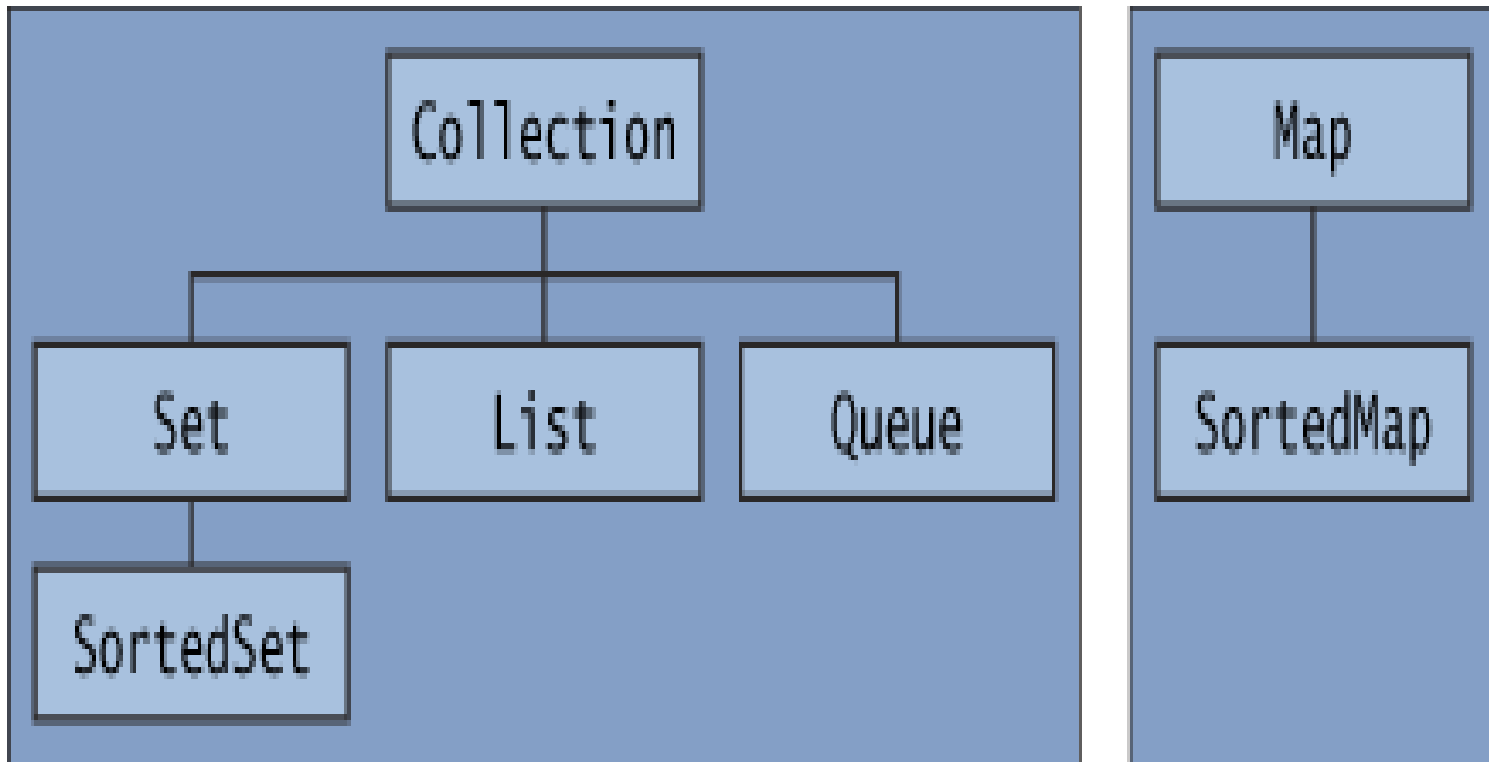
Interface **Queue**: coleção que modela uma fila de espera (primeiro elemento a entrar, primeiro elemento a sair - *FIFO*)





Interfaces da estrutura de coleções

Interface **Map**: coleção que associa chaves a valores e não pode conter chaves duplicadas





Interfaces da estrutura de coleções

- Várias implementações para essas interfaces são fornecidas dentro da estrutura de coleções (**Collections Framework**) Java
- As classes e interfaces da estrutura de coleções são membros do pacote **java.util**



Coleções com referências **Object**

- Nas primeiras versões Java, as classes na estrutura de coleções armazenavam e manipulavam referências **Object**
- Portanto, era permitido armazenar qualquer objeto em uma coleção



Coleções com referências **Object**

- Um aspecto inconveniente de armazenar referências **Object** ocorre ao recuperá-las de uma coleção
- Se um programa precisar processar um tipo específico de objeto, as referências **Object** obtidas de uma coleção em geral têm que ser convertidas no tipo apropriado.



Coleções com **Object** - Exemplo

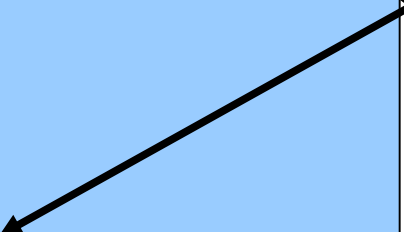
Interfaces de coleções baseadas em objetos da classe **Object** permite que as coleções agrupem qualquer tipo de objeto

```
interface List {  
    public void add(Object elemento);  
    public Object get(int indice);  
    public Iterator iterator();  
    ...  
}  
  
interface Iterator{  
    public Object next();  
    ...  
}
```

Vantagem: aceita qualquer tipo



Desvantagem: retorna tipo Object que requer coersão, verificada somente em tempo de execução





Coleções com Object - Exemplo

```
public class lista {  
    public static void main(String[] args) {  
        List list = new ArrayList();  
        Integer dado1 = new Integer(10);  
        Double dado2 = new Double(10.5);  
        for (int i = 1; i <= 50000; i++)  
            list.add(0, dado1);  
        list.add(0, dado2);  
        Iterator it = list.iterator();  
        while(it.hasNext()){  
            Integer dado3 = (Integer) it.next();  
        }  
    }  
}
```

O compilador não acusa erro, pois Integer e Double são subclasses de Object

Mensagem de erro:
Exception in thread "main"
java.lang.ClassCastException:
n: java.lang.Double



Coleções com genéricos

- A estrutura de coleções foi aprimorada com as capacidades dos genéricos
- Isso significa que é possível especificar o tipo exato que será armazenado em uma coleção
- Os benefícios da verificação de tipos em tempo de compilação também são obtidos
 - O compilador assegura que os tipos apropriados à coleção estão sendo utilizados



Coleções com genéricos

- Além disso, uma vez que o tipo armazenado em uma coleção é especificado, qualquer referência recuperada dessa coleção terá o tipo especificado
- Isso elimina a necessidade de coerções de tipo explícitas que podem lançar exceções **ClassCastException** se o objeto referenciado não for do tipo apropriado.



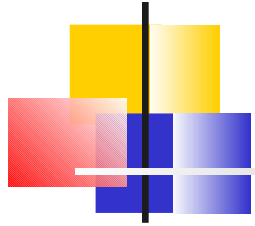
Coleções com genéricos

- Obs: os programas que foram compilados com as primeiras versões Java e que utilizam coleções podem compilar adequadamente
 - o compilador utiliza automaticamente tipos brutos ao encontrar coleções para as quais os argumentos do tipo não foram especificados



Interface **Collection**

- Uma coleção representa um grupo de objetos conhecidos como os elementos dessa coleção.
- Interface **Collection**: raiz da hierarquia de coleções
- É uma interface genérica
 - Ao declarar uma instância **Collection** deve-se especificar o tipo de objeto contido na coleção



Interface Collection

- É utilizada para manipular coleções quando deseja-se obter o máximo de generalidade
- Não garante nas implementações
 - Inexistência de duplicatas
 - Ordenação
- Fornece um objeto iterador (*Iterator*): percorre todos os elementos da coleção



Interface Collection

- Operações básicas: atuam sobre elementos individuais em uma coleção, por ex:
 - adiciona elemento (**add**)
 - remove elemento (**remove**)
- Operações de volume: atuam sobre todos os elementos de uma coleção, por ex:
 - adiciona coleção (**addAll**)
 - remove coleção (**removeAll**)
 - mantém coleção (**retainAll**)



Interface Collection

- A interface Collection também fornece operações para converter uma coleção em um array
 - `Object[] toArray()`
 - `<T> T[] toArray(T[] a)`
- Além disso, essa interface fornece um método que retorna um objeto **Iterator**:
 - permite a um programa percorrer a coleção e remover elementos da coleção durante a iteração



Interface Collection

- Outros métodos permitem:
 - determinar quantos elementos pertencem à coleção
 - **int size()**
 - determinar se uma coleção está ou não vazia
 - **boolean isEmpty()**



Classe Collection

- A classe **Collections** fornece métodos **static** que manipulam as coleções polimorficamente
- Esses métodos implementam algoritmos para:
 - busca
 - ordenação, etc



Listas

- Uma coleção **List** é uma **Collection** ordenada que pode conter elementos duplicados
- Como os arrays, os índices de uma coleção **List** são baseados em 0 (zero)
 - ié, o índice do primeiro elemento é zero



Listas

- Além dos métodos herdados de **Collection**, a interface **List** fornece métodos para:
 - manipular elementos via seus índices
 - manipular um intervalo específico de elementos
 - procurar elementos
 - obter um **ListIterator** para acessar os elementos



Listas

- A interface **List** é implementada por várias classes, incluídas as classes:
 - **ArrayList**
 - **LinkedList**
 - **Vector**



Listas

- A classe **ArrayList** e a classe **Vector** são implementações de arrays redimensionáveis da interface **List**
- A classe **LinkedList** é uma implementação de lista encadeada da interface **List**



Listas

- O comportamento e as capacidades da classe **ArrayList** são semelhantes às da classe **Vector**
- Entretanto, a classe **Vector** é do Java 1.0, antes de a estrutura de coleções ser adicionada ao Java
 - **Vector** tem vários métodos que não fazem parte da interface **List** e que não estão implementados em **ArrayList**, embora realizem tarefas idênticas



Listas

- Por exemplo, os métodos **add** e **addElement** da classe **Vector** acrescentam um elemento a um objeto **Vector**
 - mas somente o método **add** é especificado na interface **List** e implementado na classe **ArrayList**



Listas

- Objetos da classe **LinkedList** podem ser utilizados para criar:
 - pilhas
 - filas
 - deque (filas com dupla terminação)
 - árvores



ArrayList e Iterator

Exemplo

- Tarefa1: colocar dois arrays de **String** em duas listas **ArrayList**
- Tarefa 2: utilizar um objeto **Iterator** para remover da segunda coleção **ArrayList** todos os elementos que também estiverem na primeira coleção



ArrayList e Iterator

Exemplo

```
1. import java.util.List;
2. import java.util.ArrayList;
3. import java.util.Collection;
4. import java.util.Iterator;

6. public class CollectionTest
7. {
8.     private static final String[] colors =
9.         { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
10.    private static final String[] removeColors =
11.        { "RED", "WHITE", "BLUE" };
```



ArrayList e Iterator

Exemplo

```
1. // cria ArrayList, adiciona Colors a ela e a manipula
2. public CollectionTest() {
3.     List< String > list = new ArrayList< String >();
4.     List< String > removeList = new ArrayList< String >();

5.
6.     // adiciona elementos no array colors a listar
7.     for ( String color : colors )
8.         list.add( color );

9.
10.    // adiciona elementos de removeColors a removeList
11.    for ( String color : removeColors )
12.        removeList.add( color );
```



ArrayList e Iterator

Exemplo

Cria objetos **ArrayList** e atribui suas referências a variáveis

```
1. // cria ArrayList, adiciona
2. public CollectionTest() {
3.     List< String > list = new ArrayList< String >();
4.     List< String > removeList = new ArrayList< String >();

6.     // adiciona elementos no array colors a listar
7.     for ( String color : colors )
8.         list.add( color );

10.    // adiciona elementos de removeColors a removeList
11.    for ( String color : removeColors )
12.        removeList.add( color );
```

ArrayList e Iterator

Exer

Essas duas listas armazenam objetos **String**

```
1. // cria ArrayList
2. public Collection {
3.     List< String > list = new ArrayList< String >();
4.     List< String > removeList = new ArrayList< String >();

6.     // adiciona elementos no array colors a listar
7.     for ( String color : colors )
8.         list.add( color );

10.    // adiciona elementos de removeColors a removeList
11.    for ( String color : removeColors )
12.        removeList.add( color );
```



ArrayList e Iterator

Exemplo

```
1. // cria ArrayList, adiciona Colors a ela e a manipula
2. public CollectionTest() {
3.     List< String > list
4.     List< String > removeColors
5.
6.     // adiciona elementos no ArrayList colors a list
7.     for ( String color : colors )
8.         list.add( color );
9.
10.    // adiciona elementos de removeColors a removeList
11.    for ( String color : removeColors )
12.        removeList.add( color );
```

Preenche a coleção **list**
com objetos **String**
armazenados no array **color**



ArrayList e Iterator

Exemplo

```
1. // cria ArrayList, adiciona Colors a ela e a manipula
2. public CollectionTest() {
3.     List< String > list = new ArrayList< String >();
4.     List< String > removeList = new ArrayList< String >();

6.     // adiciona elementos no array colors a listar
7.     for ( String
8.         list.add(

10.    // adiciona elementos removeColors a removeList
11.    for ( String color : removeColors )
12.        removeList.add( color );
```

Preenche a coleção **removelist** com objetos **String** armazenados no array **removecolor**



ArrayList e Iterator

Exemplo

```
1. System.out.println( "ArrayList: " );

3.     // gera saída do conteúdo da lista
4.     for ( int count = 0; count < list.size(); count++ )
5.         System.out.printf( "%s ", list.get( count ) );

7.     // remove cores contidas em removeList
8.     removeColors( list, removeList );

10.    System.out.println("\n\nArrayList after calling
    removeColors: " );

12.    // gera saída do conteúdo da lista
13.    for ( String color : list )
14.        System.out.printf( "%s ", color );
15. } // fim do construtor CollectionTest
```

ArrayList e Iterator

Exe

Chama o método **get** da interface **List** para obter cada elemento da lista

```
1. System.out.  
3. // gera saída da lista  
4. for ( int count = 0; count < list.size(); count++ )  
5.     System.out.printf( "%s ", list.get( count ) );  
  
7. // remove cores contidas em removeList  
8. removeColors( list, removeList );  
  
10. System.out.println("\n\nArrayList after calling  
    removeColors: " );  
  
12. // gera saída do conteúdo da lista  
13. for ( String color : list )  
14.     System.out.printf( "%s ", color );  
15. } // fim do construtor CollectionTest
```

ArrayList e Iterator

Exemplo

Chama o método **size** da interface **List** para obter o número de elementos da lista

```
1. System.out.println( "Arr

3.         // gera saída do conteúdo da list

4.         for ( int count = 0; count < list.size(); count++ )

5.             System.out.printf( "%s ", list.get( count ) );

7.         // remove cores contidas em removeList

8.         removeColors( list, removeList );

10.        System.out.println("\n\nArrayList after calling
        removeColors: " );

12.        // gera saída do conteúdo da lista

13.        for ( String color : list )

14.            System.out.printf( "%s ", color );

15.    } // fim do construtor CollectionTest
```

ArrayList e Iterator

Exo

Uma instrução **for** aprimorada poderia ter sido utilizada aqui!!

```
1. System.out.println("ArrayList before calling  
2. removeColors: ");  
3. // gera saída do conteúdo da lista  
4. for ( int count = 0; count < list.size(); count++ )  
5.     System.out.printf( "%s ", list.get( count ) );  
6.  
7. // remove cores contidas em removeList  
8. removeColors( list, removeList );  
9.  
10. System.out.println("\n\nArrayList after calling  
    removeColors: " );  
11.  
12. // gera saída do conteúdo da lista  
13. for ( String color : list )  
14.     System.out.printf( "%s ", color );  
15. } // fim do construtor CollectionTest
```



ArrayList e Iterator

Exemplo

```
1. System.out.println( "ArrayList: " );

3.      // gera saída do conteúdo da lista
      for ( int count = 0; count < list.size(); count++ )
      {
          System.out.printf( "%s ", list.get( count ) );
          System.out.println();
      }

7.      // remove cores contidas em removeList
8.      removeColors( list, removeList );

10.     System.out.println("\n\nArrayList after calling
        removeColors: " );

12.     // gera saída do conteúdo da lista
13.     for ( String color : list )
14.         System.out.printf( "%s ", color );
15. } // fim do construtor CollectionTest
```

Chamada do método
removeColors

ArrayList e Iterator

Exemplo

Remove de **collection2** as cores
(objetos **String**) especificadas em **collection1**

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext())  
  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
    } // fim do método removeColors
```

ArrayList e Iterator

Exemplo

Permite que quaisquer objetos **Collections** que conttenham strings sejam passados como argumentos

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext())  
  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
    } // fim do método removeColors
```

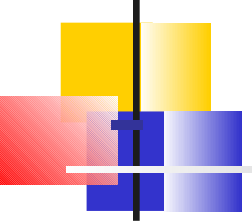
Arrendizagem de Iteradores

Ex

O método acessa os elementos da primeira coleção via um **Iterator**.
Chama o método **iterator** para obter um iterador para **collection1**

```
private void removeColors(Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext())  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
    } // fim do método removeColors
```


ArrayList e Iterator



Observe que os tipos **Collection** e **Iterator** são genéricos!!

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext())  
  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
    } // fim do método removeColors
```



ArrayList e Iterator

Exemplo

```
private void removeColors( Collection< String > collection1,
```

Chama o método **hasnext** da classe **Iterator** para determinar se a coleção tem mais elementos

```
    // loop enquanto coleção tiver itens  
    while (iterator.hasNext())
```

```
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
    } // fim do método removeColors
```



ArrayList e Iterator

Exemplo

```
private void removeColors( Collection< String > collection1,
```

O método **hasnext** retorna **true**
se outro elemento existir e **false** caso contrário

```
// loop enquanto coleção tiver itens  
while (iterator.hasNext())
```

```
    if (collection2.contains( iterator.next() ))  
        iterator.remove();// remove Color atual  
} // fim do método removeColors
```



ArrayList e Iterator

Exemplo

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){
```

```
    // obtém o iterador
```

Chama método **next** da classe **Iterator**
para obter uma referência ao próximo
elemento da coleção

```
    while (collection2.iterator().hasNext())
```

```
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual
```

```
    } // fim do método removeColors
```



ArrayList e Iterator

Exemplo

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){
```

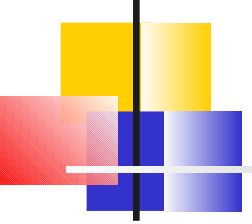
```
    // obtém o iterador
```

Utiliza o método **contains** da segunda coleção para determinar se a mesma contém o elemento retornado por **next**

```
    while ( ! iterator.hasNext() )
```

```
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual
```

```
    } // fim do método removeColors
```



Erro de programação comum

- Se uma coleção for modificada por um de seus métodos depois de um iterador ter sido criado para essa coleção:
 - o iterador se torna imediatamente inválido!



LinkedList - Exemplo

```
1.  import java.util.List;
2.  import java.util.LinkedList;
3.  import java.util.ListIterator;
4.  public class ListTest
5.  {
6.      private static final String colors[] = { "black",
"yellow", "green", "blue", "violet", "silver" };
7.      private static final String colors2[] = { "gold", "white",
"brown", "blue", "gray", "silver" };
8.
9.      // configura e manipula objetos LinkedList
10.     public ListTest()
11.     {
12.         List< String > list1 = new LinkedList< String >();
13.         List< String > list2 = new LinkedList< String >();
```



LinkedList - Exemplo

```
public class ListTest
{
    private static final String colors[] = { "black", "yellow",
        "green", "blue", "violet", "silver" };
    private static final String colors2[] = { "gold", "white",
```

Cria duas listas **LinkedList**
contendo elementos **String**

```
};
```

```
public void Test()
```

```
{
```

```
    List< String > list1 = new LinkedList< String >();
```

```
    List< String > list2 = new LinkedList< String >();
```




LinkedList - Exemplo

Adiciona elementos às duas listas

```
■ // adiciona elementos a list1
■   for ( String color : colors )
■       list1.add( color );

■ // adiciona elementos a list2
■   for ( String color : colors2 )
■       list2.add( color );
```



LinkedList - Exemplo

Todos elementos da lista **list1** são adicionados à lista **list2**

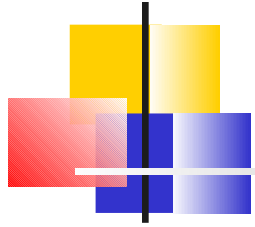
- `list1.addAll(list2); // concatena as listas`
- `printList(list1); // imprime elementos list1`



LinkedList - Exemplo

Chama o método **addAll**
da classe **List**

- `list1.addAll(list2); // concatena as listas`
- `printList(list1); // imprime elementos list1`



LinkedList - Exemplo

- `list1.addAll(list2); // concatena as listas`
- `printList(list1); // imprime elementos list1`

Chama o método **printlist** para gerar a saída do conteúdo de **list1**



LinkedList - Exemplo

Gera saída do conteúdo de **List**

```
public void printList(List< String > list)
{
    System.out.println( "\nlist: " );

    for ( String color : list )
        System.out.printf( "%s ", color );

    System.out.println();
} // fim do método printList
```



LinkedList - Exemplo

Converte cada elemento **string** da lista em letras maiúsculas

1. `convertToUppercaseStrings(list1);`
2. `printList(list1); // imprime elementos list1`

Chama o método **printlist** para gerar a saída do conteúdo de **list1**



LinkedList - Exemplo

Localiza objetos **String** e
converte em letras maiúsculas

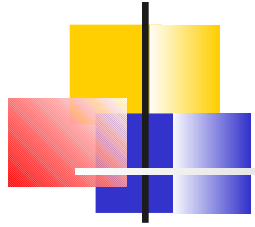
```
private void convertToUppercaseStrings(List< String > list){  
    ListIterator< String > iterator = list.listIterator();  
  
    while (iterator.hasNext())  
    {  
        String color = iterator.next(); // obtém o item  
        iterator.set( color.toUpperCase() ); // converte em  
        letras maiúsculas  
    } // fim do while  
} // fim do método convertToUppercaseStrings
```



LinkedList - Exemplo

Chama o método **listIterator** da interface **List** para obter um iterador bidirecional para a lista

```
private void convertToUppercaseStrings(List< String > list){  
    ListIterator< String > iterator = list.listIterator();  
  
    while (iterator.hasNext())  
    {  
        String color = iterator.next(); // obtém o item  
        iterator.set( color.toUpperCase() ); // converte em  
        letras maiúsculas  
    } // fim do while  
} // fim do método convertToUppercaseStrings
```

LinkedList - Exemplo

```
private void convertToUppercaseStrings(List< String > list){
```

Chama o método **toUpperCase** da classe **String** para obter uma versão em letras maiúsculas da **string**

```
{  
    String color = iterator.next(); // obtém o item  
    iterator.set( color.toUpperCase() ); // converte em  
    letras maiúsculas  
} // fim do while  
} // fim do método convertToUppercaseStrings
```



LinkedList - Exemplo

`private`

Chama o método **set** da classe **ListIterator** para substituir a **string** referenciada pelo iterador pela sua versão em letras maiúsculas

```
{  
    String = iterator.next(); // obtém o item  
    iterator.set( color.toUpperCase() ); // converte em  
    letras maiúsculas  
} // fim do while  
} // fim do método convertToUppercaseStrings
```



LinkedList - Exemplo

Chama o método **removeItems** para remover os elementos que iniciam no índice 4 até, mas não incluindo o índice 7 da lista

```
1. System.out.println( "\nRemovendo os elementos de 4 a 6..." );  
2. removeItems( list1, 4, 7 );  
3. printList( list1 ); // imprime elementos list1
```

Chama o método **printlist** para gerar a saída do conteúdo de **list1**



LinkedList - Exemplo

Obtém sublista e utiliza método **clear** para excluir itens da sublista

```
private void removeItems(List< String > list, int start, int end)
{
    list.subList( start, end ).clear(); // remove os itens
} // fim do método removeItems
```



LinkedList - Exemplo

Chama o método **sublist** da classe **List** para obter uma parte da lista

```
private void removeItems(LinkedList list, int start, int end)
{
    list.subList( start, end ).clear(); // remove os itens
} // fim do método removeItems
```



LinkedList - Exemplo

O método **sublist** aceita dois argumentos:
os índices inicial e final da sublista

Obs: o índice final não faz parte da sublista!

```
private void removeItems(int start, int end)
{
    list.subList( start, end ).clear(); // remove os itens
} // fim do método removeItems
```



LinkedList - Exemplo

Chama o método **clear** da classe **List** para remover todos os elementos da sublista contida na lista **list**

```
private void removeItems(int start, int end)
{
    list.subList( start, end ).clear(); // remove os itens
} // fim do método removeItems
```



LinkedList - Exemplo

Imprime a lista na ordem inversa

```
1. printReversedList( list1 );
```




LinkedList - Exemplo

Imprime a lista invertida
(de trás pra frente)

```
private void printReversedList(List< String > list){  
    ListIterator< String > iterator = list.listIterator( list.size() );  
  
    System.out.println( "\nReversed List:" );  
  
    // imprime lista na ordem inversa  
    while (iterator.hasPrevious())  
        System.out.printf( "%s ", iterator.previous());  
} // fim do método printReversedList
```



LinkedList - Exemplo

Chama o método **listIterator** da classe **List** com um argumento que especifica a posição inicial do iterador (nesse caso, o último elemento)

```
private void printReversedList(List<String> list){  
    ListIterator<String> iterator = list.listIterator( list.size() );  
  
    System.out.println( "\nReversed List:" );  
  
    // imprime lista na ordem inversa  
    while (iterator.hasPrevious())  
        System.out.printf( "%s ", iterator.previous());  
} // fim do método printReversedList
```



LinkedList - Exemplo

Chama o método **hasprevious** da classe **ListIterator** para determinar se há mais elementos ao percorrer a lista em ordem invertida

```
// imprime lista na ordem inversa
while (iterator.hasPrevious())
    System.out.printf( "%s ", iterator.previous());
} // fim do método printReversedList
```

```
st.size() );
```



LinkedList - Exemplo

```
private void printReversedList(List< String > list){  
    ListIterator< String> iterator = list.listIterator()  
  
    System.out.println("Imprimindo lista na ordem inversa")  
  
    // imprime lista na ordem inversa  
    while (iterator.hasPrevious())  
        System.out.printf( "%s ", iterator.previous());  
} // fim do método printReversedList
```

Chama o método **previous** da classe **ListIterator** para obter o elemento anterior da lista



Algoritmos de coleções

- A estrutura de coleções fornece vários algoritmos para operações em coleções
- Esses algoritmos estão implementados como métodos **static** da classe **Collections**



Algoritmos de coleções

- Algoritmos que operam em objetos da classe **List**:
 - **sort**: classifica os elementos da lista
 - **binarySearch**: localiza um elemento da lista
 - **reverse**: inverte os elementos da lista
 - **shuffle**: "embaralha" os elementos da lista
 - **fill**: preenche uma lista
 - **copy**: copia referências de uma lista em outra lista



Algoritmos de coleções

- Algoritmos que operam em objetos da classe **Collection**:
 - **min**: retorna o menor elemento em uma coleção
 - **max**: retorna o maior elemento em uma coleção
 - **frequency**: calcula quantos elementos em uma coleção são iguais a um elemento especificado
 - **disjoint**: determina se duas coleções não têm nenhum elemento em comum



Algoritmo Sort

- O algoritmo **sort** classifica (ordena) os elementos de uma lista **List**
 - os tipos dos elementos da lista devem implementar a interface **Comparable**
- A ordem entre os elementos da lista é determinada pela ordem natural do tipo dos elementos
 - esquema implementado pela classe no método **compareTo**



Algoritmo Sort - Exemplo

Classifica os elementos de
uma lista em ordem crescente

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1 {
    private static final String suits[] =
        { "Hearts", "Diamonds", "Clubs", "Spades" };

    public void printElements(){ // exhibe elementos do array
        List< String > list = Arrays.asList( suits ); // cria List

        // gera saída da lista
        System.out.printf( "Unsorted array elements:\n%s\n", list);

        Collections.sort( list ); // classifica ArrayList

        // gera saída da lista
        System.out.printf( "Sorted array elements:\n%s\n", list);
    } // fim do método printElements
```



Algoritmo Sort - Exemplo

Chama o método **asList** da classe **Arrays** para permitir que o conteúdo do array seja manipulado como uma lista

```
import java.util.*;
import java.io.*;
import java.awt.*;

public class Sort {
    private static String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };

    public void printElements() { // exibe elementos do array
        List< String > list = Arrays.asList( suits ); // cria List

        // gera saída da lista
        System.out.printf( "Unsorted array elements:\n%s\n", list);

        Collections.sort( list ); // classifica ArrayList

        // gera saída da lista
        System.out.printf( "Sorted array elements:\n%s\n", list);
    } // fim do método printElements
}
```



Algoritmo Sort - Exemplo

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1 {
    private static final S
        { "Hearts", "Diamond"

    public void printElements() {
        List< String > list = Arrays.asList( "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K" ); // cria List

        // gera saída da lista
        System.out.printf( "Unsorted array elements:\n%s\n", list);

        Collections.sort( list ); // classifica ArrayList

        // gera saída da lista
        System.out.printf( "Sorted array elements:\n%s\n", list);
    } // fim do método printElements
}
```

Chamada implícita ao método **toString** da classe **List** para gerar a saída do conteúdo da lista



Algoritmo Sort - Exemplo

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

public class Sort1 {
    private static final String suits[] =
        { "Hearts", "Diamonds", "Clubs", "Spades" };

    public void printElements( elementos do array
        List<String> list = Arrays.asList( suits ); // cria List

        // gera saída da lista não classificada
        System.out.printf( "Unsorted array elements:\n%s\n", list);

        Collections.sort( list ); // classifica ArrayList

        // gera saída da lista classificada
        System.out.printf( "Sorted array elements:\n%s\n", list);
    } // fim do método printElements
```

Classifica a lista **list**
em ordem crescente



Algoritmo **BinarySearch**

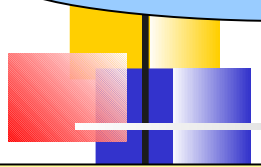
- O algoritmo **BinarySearch** tenta localizar um elemento em uma lista **List**
 - se o elemento for encontrado, seu índice é retornado
 - se o elemento não for encontrado, o algoritmo retorna um valor negativo



Algoritmo BinarySearch

- Observação: o algoritmo **BinarySearch** espera que os elementos da lista estejam classificados em ordem crescente

Utiliza o algoritmo **BinarySearch** para procurar uma série de strings em uma **ArrayList**



```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest
{
    private static final String colors[] = { "red", "white",
        "blue", "black", "yellow", "purple", "tan", "pink" };
    private List< String > list; // ArrayList reference

    // cria, classifica e gera a saída da lista
    public BinarySearchTest()
    {
        list = new ArrayList< String >( Arrays.asList( colors ) );
        Collections.sort( list ); // sort the ArrayList
        System.out.printf( "Sorted ArrayList: %s\n", list );
    } // end BinarySearchTest constructor
```

Algoritmo BinarySearch -

Exemplo

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest
{
    private static final String colors[] = { "red", "white",
        "blue", "black", "yellow", "purple", "tan", "pink" };

    // Classifica a lista com o método sort
    // da interface Collection

    public BinarySearchTest()
    {
        list = new ArrayList< String >( Arrays.asList( colors ) );
        Collections.sort( list ); // sort the ArrayList
        System.out.printf( "Sorted ArrayList: %s\n", list );
    } // end BinarySearchTest constructor
}
```


Algoritmo BinarySearch -

Exemplo

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;
import java.util.ArrayList;

public class BinarySearchTest
{
    private static final String colors[] = { "red", "white",
        "blue", "black", "yellow", "purple", "tan", "pink" };
    private List< String > list; // ArrayList reference

    // constructor
    public BinarySearchTest()
    {
        list = new ArrayList< String >( Arrays.asList( colors ) );
        Collections.sort( list ); // sort the ArrayList
        System.out.printf( "Sorted ArrayList: %s\n", list );
    } // end BinarySearchTest constructor
```

Algoritmo BinarySearch -

Exemplo

Realiza a pesquisa e envia o resultado para a saída

```
private void printSearchResults( String key )
{
    int result = 0;

    System.out.printf( "\n Buscando o elemento: %s\n", key );
    result = Collections.binarySearch( list, key );

    if ( result >= 0 )
        System.out.printf( "Elemento encontrado no índice %d\n", result );
    else
        System.out.printf( "Elemento não encontrado! (%d)\n", result );
} // fim do método printSearchResults
```

Algoritmo **BinarySearch** -

Exemplo

Chama o método **binarySearch** da interface **Collection** para buscar um elemento (**key**) em uma lista (**list**)

```
private void printSearchResults( String key )
{
    int result = 0;

    System.out.printf( "\n Buscando o elemento: %s\n", key );
    result = Collections.binarySearch( list, key );

    if ( result >= 0 )
        System.out.printf( "Elemento encontrado no índice %d\n", result );
    else
        System.out.printf( "Elemento não encontrado! (%d)\n", result );
} // fim do método printSearchResults
```



Conjuntos

- Um conjunto (**Set**) é uma coleção que contém elementos únicos não duplicados
- A classe **HashSet** implementa a interface **Set**

HashSet - Exemplo

Exemplo de um método que aceita um argumento **Collection** e constrói uma coleção **HashSet** a partir desse argumento

```
private void printNonDuplicates( Collection< String > collection )
{
    // create a HashSet
    Set< String > set = new HashSet< String >( collection );

    System.out.println( "\n A coleção sem duplicatas: " );

    for ( String s : set )
        System.out.printf( "%s ", s );

    System.out.println();
} // end method printNonDuplicates
```

HashSet - Exemplo

Por definição, coleções da classe **Set** não contêm duplicatas, então quando a coleção **HashSet** é construída, ela remove quaisquer duplicatas passadas pelo argumento **Collection**

```
private void printNonDuplicates( Collection< String > collection )
{
    // create a HashSet
    Set< String > set = new HashSet< String >( collection );

    System.out.println( "\n A coleção sem duplicatas: " );

    for ( String s : set )
        System.out.printf( "%s ", s );

    System.out.println();
} // end method printNonDuplicates
```

HashSet - Exemplo

Qual é a saída desse programa?

```
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class SetTest {
    private static final String colors[] = { "vermelho", "branco", "azul",
        "verde", "cinza", "laranja", "amarelo", "branco", "rosa",
        "violeta", "cinza", "laranja" };

    public SetTest(){
        List< String > list = Arrays.asList( colors );
        System.out.printf( "ArrayList: %s\n", list );
        printNonDuplicates( list );
    } // end SetTest constructor

    public static void main( String args[] ){
        new SetTest();
    } // end main
} // end class SetTest
```



HashSet - Exemplo

■ Saída do programa:

```
ArrayList: [vermelho, branco, azul, verde, cinza, laranja, amarelo,  
branco, rosa, violeta, cinza, laranja]
```

A coleção sem duplicatas:

```
vermelho branco azul verde cinza laranja amarelo rosa violeta
```

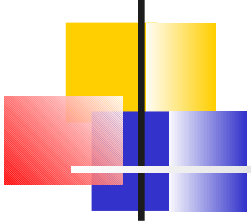



Conjuntos ordenados

- A estrutura de coleções também inclui a interface **SortedSet**:
 - estende a interface **Set**
 - representa conjuntos que mantêm seus elementos ordenados
- A classe **TreeSet** implementa a interface **SortedSet**

Coleção TreeSet -

Exemplo



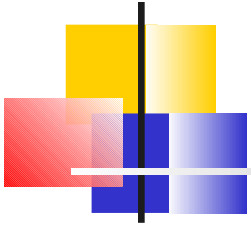
Inserir strings em uma coleção TreeSet

```
1.  import java.util.Arrays;
2.  import java.util.SortedSet;
3.  import java.util.TreeSet;

5.  public class SortedSetTest
6.  {
7.      private static final String names[] = { "amarelo", "verde",
8.          "preto", "marrom", "cinza", "branco", "laranja", "vermelho", "verde" };
9.
10.     // create a sorted set with TreeSet, then manipulate it
11.     public SortedSetTest()
12.     {
13.         // create TreeSet
14.         SortedSet< String > tree =
15.             new TreeSet< String >( Arrays.asList( names ) );
```

Coleção TreeSet -

Exemplo



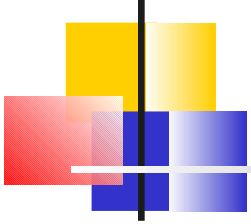
```
1.  import java.util.Arrays;
2.  import java.util.SortedSet;
3.  import java.util.TreeSet;

5.  public class SortedSetTest
6.  {
7.      private static final String[] names = { "John", "Mary", "Bob", "Alice", "David", "Eve", "Frank", "Grace", "Henry", "Ivy", "Jack", "Karen", "Leo", "Mia", "Noah", "Olivia", "Peter", "Quinn", "Ryan", "Sophia", "Tyler", "Uma", "Victor", "Wendy", "Xavier", "Yara", "Zoe" };
8.
9.
10.
11.
12.  {
13.      // create TreeSet
14.      SortedSet< String > tree =
15.          new TreeSet< String >( Arrays.asList( names ) );
```

As strings são classificadas à medida em que são adicionadas à coleção **TreeSet**

Coleção TreeSet -

Exemplo



```
1. System.out.println( "conjunto ordenado: " );  
2. printSet( tree );
```

Gera a saída do conjunto inicial de strings utilizando o método **printSet**

Coleção TreeSet -

Exemplo



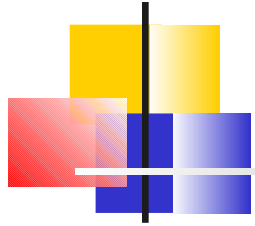
Aceita uma coleção **SortedSet** como argumento e a imprime

```
private void printSet( SortedSet< String > set )
{
    for ( String s : set )
        System.out.print( s + " " );

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

Coleção TreeSet -

Exemplo



```
1. System.out.println( "conjunto ordenado: " );  
2. printSet( tree );
```

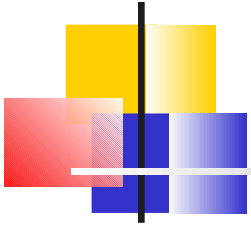
Saída:

conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

Coleção TreeSet -

Exemplo

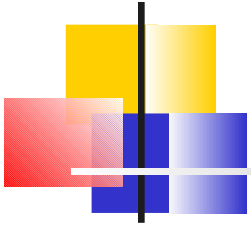


```
1. System.out.print( "\n headSet (\"laranja\"):  " );  
2. printSet( tree.headSet( "laranja" ) );
```

Chama o método **headSet** da classe **TreeSet** para obter um subconjunto em que cada elemento é maior do que **“laranja”**

Coleção TreeSet -

Exemplo



```
1. System.out.print( "\n headSet (\"laranja\"):  " );  
2. printSet( tree.headSet( "laranja" ) );
```

Saída:

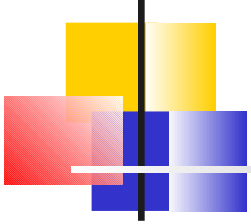
conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

headSet("laranja"): amarelo branco cinza

Coleção TreeSet -

Exemplo

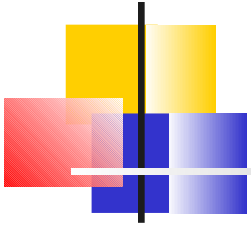


```
1. System.out.print( "\n tailSet (\"laranja\"):  " );  
2. printSet( tree.tailSet( "laranja" ) );
```

Chama o método **headSet** da classe **TreeSet** para obter um subconjunto em que cada elemento é menor ou igual a “**laranja**”

Coleção TreeSet -

Exemplo



```
1. System.out.print( "\n tailSet (\"laranja\"):  " );  
2. printSet( tree.tailSet( "laranja" ) );
```

Saída:

conjunto ordenado:

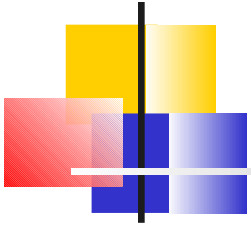
amarelo branco cinza laranja marrom preto verde vermelho

headSet("laranja"): amarelo branco cinza

tailSet("laranja"): laranja marrom preto verde vermelho

Coleção TreeSet -

Exemplo

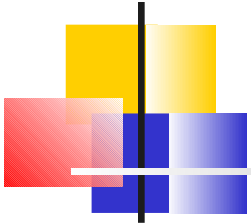


```
1. System.out.printf( "first: %s\n", tree.first() );  
2. System.out.printf( "last : %s\n", tree.last() );
```

Chama os métodos **first** e **last** da classe **TreeSet** para obter o menor e o maior elementos do conjunto

Coleção TreeSet -

Exemplo



```
1. System.out.printf( "first: %s\n", tree.first() );  
2. System.out.printf( "last : %s\n", tree.last() );
```

Saída:

conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

headSet(Í aranja) : amarelo branco cinza

tailSet(Í aranja) : laranja marrom preto verde vermelho

first: amarelo

last: vermelho



Mapas

- Um mapa (**Map**) associa chaves a valores e não pode conter chaves duplicatas
 - cada chave pode mapear somente um valor (mapeamento um para um)
- Classes que implementam a interface **Map**
 - **HashTable**
 - **HashMap**
 - **TreeMap**



Mapas ordenados

- A interface (**SortedMap**) estende a interface **Map** e mantém as suas chaves ordenadas
- A classe **TreeMap** implementa a interface **SortedMap**

Coleção HashMap -

Exemplo



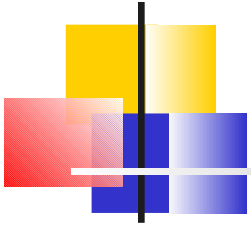
Utiliza uma coleção **HashMap** para contar o número de ocorrências de cada palavra em uma string

```
1.  import java.util.StringTokenizer;
2.  import java.util.Map;
3.  import java.util.HashMap;
4.  import java.util.Set;
5.  import java.util.TreeSet;
6.  import java.util.Scanner;

8.  public class WordTypeCount{
9.      private Map< String, Integer > map;
10.     private Scanner scanner;
11.     public WordTypeCount(){
12.         map = new HashMap< String, Integer >(); // cria HashMap
13.         scanner = new Scanner( System.in ); // cria scanner
14.         createMap(); // cria mapa baseado na entrada do usuário
15.         displayMap(); // apresenta conteúdo do mapa
16.     } //fim do construtor de WordTypeCount
```

Coleção HashMap -

Exemplo



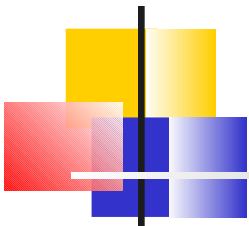
```
1. import java.util.StringTokenizer;
2. import java.util.Map;
3. import java.util.HashMap;
4. import java.util.Set;
5. import java.util.TreeSet;
6. import java.util.Scanner;
```

Cria uma coleção **HashMap** vazia com chaves do tipo **String** e valores do tipo **Integer**

```
8.
9.
10. private WordTypeCount() {
11.     public WordTypeCount() {
12.         map = new HashMap< String, Integer >(); // cria HashMap
13.         scanner = new Scanner( System.in ); // cria scanner
14.         createMap(); // cria mapa baseado na entrada do usuário
15.         displayMap(); // apresenta conteúdo do mapa
16.     } // fim do construtor de WordTypeCount
```


Coleção HashMap -

Exemplo



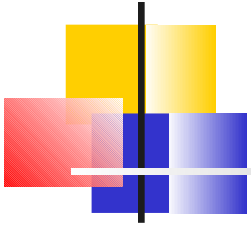
```
1.  import java.util.StringTokenizer;
2.  import java.util.Map;
3.  import java.util.HashMap;
4.  import java.util.Set;
5.  import java.util.TreeSet;
6.  import java.util.Scanner;
```

Cria uma objeto **Scanner** que lê a entrada do usuário a partir do fluxo de entrada padrão

```
11.
12.      map = new HashMap< String, Integer >(); // cria HashMap
13.      scanner = new Scanner( System.in ); // cria scanner
14.      createMap(); // cria mapa baseado na entrada do usuário
15.      displayMap(); // apresenta conteúdo do mapa
16.  } // fim do construtor de WordTypeCount
```

Coleção HashMap -

Exemplo



```
1. import java.util.StringTokenizer;
2. import java.util.Map;
3. import java.util.HashMap;
4. import java.util.Set;
5. import java.util.TreeSet;
6. import java.util.Scanner;
```

```
8.
```

Chama o método **createMap** para armazenar no mapa o número de ocorrências de cada palavra na frase

```
12.         hashMap<String, Integer>(); // cria hashMap
13.         scanner = new Scanner( System.in ); // cria scanner
14.         createMap(); // cria mapa baseado na entrada do usuário
15.         displayMap(); // apresenta conteúdo do mapa
16.     } // fim do construtor de WordTypeCount
```

Coleção HashMap -

Método **createMap**: armazena em um mapa o número de ocorrências de cada palavra na frase do usuário

```
private void createMap(){
    System.out.println( "Entre com uma string:" ); // prompt para entrada do
    usuário
    String input = scanner.nextLine();

    // cria um objeto StringTokenizer para a entrada
    StringTokenizer tokenizer = new StringTokenizer( input );

    // processando o texto de entrada
    while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada
    {
        String word = tokenizer.nextToken().toLowerCase(); // captura palavra

        // se o mapa contiver a palavra
        if ( map.containsKey( word ) ) // a palavra está no mapa?
        {
            int count = map.get( word ); // obtém contagem atual
            map.put( word, count + 1 ); // incrementa a contagem
        } // fim if
        else
            map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
    } // end while
} // end method createMap
```

Coleção HashMap -

Invoca o método **nextLine** da classe **Scanner** para ler a entrada do usuário

```
private void createMap(){
    System.out.println( "Entre com uma frase:" ); // prompt para entrada do usuário
    String input = scanner.nextLine();

    // cria um objeto StringTokenizer para a entrada
    StringTokenizer tokenizer = new StringTokenizer( input );

    // processando o texto de entrada
    while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada
    {
        String word = tokenizer.nextToken().toLowerCase(); // captura palavra

        // se o mapa contiver a palavra
        if ( map.containsKey( word ) ) // a palavra está no mapa?
        {
            int count = map.get( word ); // obtém contagem atual
            map.put( word, count + 1 ); // incrementa a contagem
        } // fim if
        else
            map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
    } // end while
} // end method createMap
```

Coleção HashMap -

Exemplo

Cria um objeto **StringTokenizer** para dividir a string de entrada em suas palavras componentes individuais

```
// cria um objeto StringTokenizer para a entrada
StringTokenizer tokenizer = new StringTokenizer( input );

// processando o texto de entrada
while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada
{
    String word = tokenizer.nextToken().toLowerCase(); // captura palavra

    // se o mapa contiver a palavra
    if ( map.containsKey( word ) ) // a palavra está no mapa?
    {
        int count = map.get( word ); // obtém contagem atual
        map.put( word, count + 1 ); // incrementa a contagem
    } // fim if
    else
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
} // end while
} // end method createMap
```

Coleção HashMap -

Exemplo

Utiliza o método **hasMoreTokens** para determinar se há mais tokens na string sendo separada em tokens

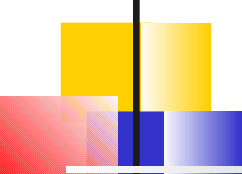
```
String input = ...;
StringTokenizer tokenizer = new StringTokenizer( input );

// processando o texto de entrada
while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada
{
    String word = tokenizer.nextToken().toLowerCase(); // captura palavra

    // se o mapa contiver a palavra
    if ( map.containsKey( word ) ) // a palavra está no mapa?
    {
        int count = map.get( word ); // obtém contagem atual
        map.put( word, count + 1 ); // incrementa a contagem
    } // fim if
    else
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
} // end while
} // end method createMap
```

Coleção HashMap -

Exemplo



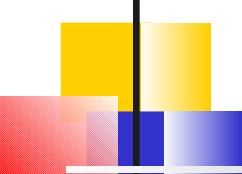
```
private void createMap(){  
    System.out.println( "Entre com uma frase:" ); // prompt para entrada do usuário  
    String frase = scanner.nextLine();  
    StringTokenizer tokenizer = new StringTokenizer( frase );  
    while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada  
    {  
        String word = tokenizer.nextToken().toLowerCase(); // captura palavra  
  
        // se o mapa contiver a palavra  
        if ( map.containsKey( word ) ) // a palavra está no mapa?  
        {  
            int count = map.get( word ); // obtém contagem atual  
            map.put( word, count + 1 ); // incrementa a contagem  
        } // fim if  
        else  
            map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa  
    } // end while  
} // end method createMap
```

Se houver mais tokens, o próximo é convertido em letras minúsculas

```
while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada  
{  
    String word = tokenizer.nextToken().toLowerCase(); // captura palavra  
  
    // se o mapa contiver a palavra  
    if ( map.containsKey( word ) ) // a palavra está no mapa?  
    {  
        int count = map.get( word ); // obtém contagem atual  
        map.put( word, count + 1 ); // incrementa a contagem  
    } // fim if  
    else  
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa  
} // end while  
} // end method createMap
```

Coleção HashMap -

Exemplo



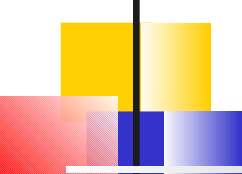
```
private void createMap(){  
    System.out.println( "Entre com uma frase:" ); // prompt para entrada do usuário
```

O próximo token é obtido com uma chamada ao método **nextToken** da classe **StringTokenizer**, que retorna uma string

```
while ( tokenizer.hasMoreTokens() ) // enquanto tiver mais entrada  
{  
    String word = tokenizer.nextToken().toLowerCase(); // captura palavra  
  
    // se o mapa contiver a palavra  
    if ( map.containsKey( word ) ) // a palavra está no mapa?  
    {  
        int count = map.get( word ); // obtém contagem atual  
        map.put( word, count + 1 ); // incrementa a contagem  
    } // fim if  
    else  
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa  
} // end while  
} // end method createMap
```


Coleção HashMap -

Exemplo



```
private void createMap(){
    System.out.println( "Entre com uma frase:" ); // prompt para entrada do usuário
    String input = scanner.nextLine();

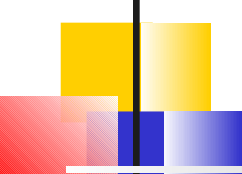
    // cria um objeto StringTokenizer para a entrada
```

Chama o método **containsKey** da classe **Map** para determinar se a palavra já está no mapa

```
        // se o mapa contiver a palavra
        if ( map.containsKey( word ) ) // a palavra está no mapa?
        {
            int count = map.get( word ); // obtém contagem atual
            map.put( word, count + 1 ); // incrementa a contagem
        } // fim if
    else
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
} // end while
} // end method createMap
```

Coleção HashMap -

Exemplo



```
private void createMap(){
    System.out.println( "Entre com uma frase:" ); // prompt para entrada do usuário
    String input = scanner.nextLine();

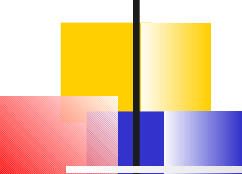
    // cria um objeto StringTokenizer para a entrada
    StringTokenizer tokenizer = new StringTokenizer( input );
```

Se a palavra estiver no mapa, utiliza o método **get** da classe **Map** para obter o valor associado (a contagem) da chave no mapa

```
        if ( map.containsKey( word ) ) // a palavra está no mapa?
        {
            int count = map.get( word ); // obtém contagem atual
            map.put( word, count + 1 ); // incrementa a contagem
        } // fim if
    else
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
    } // end while
} // end method createMap
```

Coleção HashMap -

Exemplo



```
private void createMap(){
    System.out.println( "Entre com uma frase:" ); // prompt para entrada do usuário
    String input = scanner.nextLine();

    // cria um objeto StringTokenizer para a entrada
    StringTokenizer tokenizer = new StringTokenizer( input );

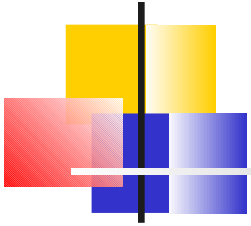
    // processando o texto de entrada
    while (tokenizer.hasMoreTokens()) {
        String word = tokenizer.nextToken();

        // Incrementa esse valor e utiliza o método
        // put da classe Map para substituir o valor
        // associado à chave no mapa
        int count = map.get( word ); // obtém contagem atual
        map.put( word, count + 1 ); // incrementa a contagem
    } // fim if
    else
        map.put( word, 1 ); // adiciona a nova palavra com contagem 1 ao mapa
    } // end while
} // end method createMap
```

trada
ura palavra
mapa?

Coleção HashMap -

Exemplo



```
1. import java.util.StringTokenizer;
2. import java.util.Map;
3. import java.util.HashMap;
4. import java.util.Set;
5. import java.util.TreeSet;
6. import java.util.Scanner;
```

```
8. public class WordTypeCount{
```

Chama o método **displayMap** para exibir todas as entradas do mapa

```
13. Scanner scanner = new Scanner(System.in); // cria scanner
14. Map<String, Integer> mapa = new HashMap<>(); // cria mapa baseado na entrada do usuário
15. displayMap(); // apresenta conteúdo do mapa
16. } // fim do construtor de WordTypeCount
```

Coleção HashMap -

Método **displayMap**: exibe todas as entradas armazenadas em um mapa

Exemplo

```
private void displayMap()
{
    Set< String > keys = map.keySet(); // obtém as chaves

    // ordena as chaves
    TreeSet< String > sortedKeys = new TreeSet< String >( keys );

    System.out.println( "O mapa contém:\nKey\t\tValue" );

    // gera a saída para cada chave no mapa
    for ( String key : sortedKeys )
        System.out.printf( "%s\t\t%s \n", key, map.get( key ) );
} // end method displayMap
```

Coleção HashMap -

Utiliza o método **keySet** da classe **HashMap** para obter um conjunto das chaves

```
private void displayMap()
{
    Set< String > keys = map.keySet(); // obtém as chaves

    // ordena as chaves
    TreeSet< String > sortedKeys = new TreeSet< String >( keys );

    System.out.println( "O mapa contém:\nKey\t\tValue" );

    // gera a saída para cada chave no mapa
    for ( String key : sortedKeys )
        System.out.printf( "%s\t\t%s\t\t\n", key, map.get( key ) );
} // end method displayMap
```

Coleção HashMap -

Exemplo

private

Cria um objeto **TreeSet** com as chaves,
em que as chaves são ordenadas

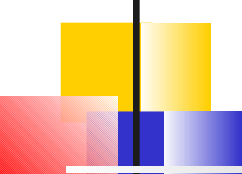
```
// ordena as chaves
TreeSet< String > sortedKeys = new TreeSet< String >( keys );

System.out.println( "O mapa contém:\nKey\t\tValue" );

// gera a saída para cada chave no mapa
for ( String key : sortedKeys )
    System.out.printf("%s\t\t%s \n", key, map.get( key ) );
} // end method displayMap
```

Coleção HashMap -

Exemplo



```
private void displayMap()
{
    Set< String > keys = map.keySet(); // obtém as chaves

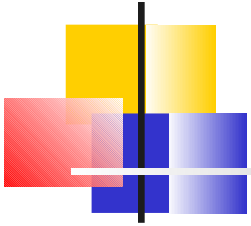
    // ordena as chaves

    for ( String key : keys )
    {
        // imprime cada chave e seu valor no mapa
        System.out.printf("%s\t\t%s \n", key, map.get( key ) );
    } // end method displayMap
```

Imprime cada chave e seu valor no mapa

Coleção HashMap -

Exemplo



Saída:

Entre com uma frase:

To be or not to be: that is the question

0 mapa contém:

Key	Value
be	1
be:	1
is	1
not	1
or	1
question	1
that	1
the	1
to	2