

Computação Orientada a Objetos

Coleções Java Parte IV

Slides baseados em:

Deitel, H.M.; Deitel P.J. **Java: Como Programar**, Pearson
Prentice Hall, 6a Edição, 2005. **Capítulo 19**

Profa. Karina Valdivia Delgado
EACH-USP

Revisando: O que é uma coleção?

- É uma estrutura de dados (um objeto) que agrupa referências a vários outros objetos.

Interfaces da estrutura de coleções

*Coleções de
elementos individuais*

*java.util.**Collection***



*java.util.**Queue***

primeiro em
entrar, primeiro
em sair

*java.util.**List***

- *seqüência definida*
- *elementos indexados*

*java.util.**Set***

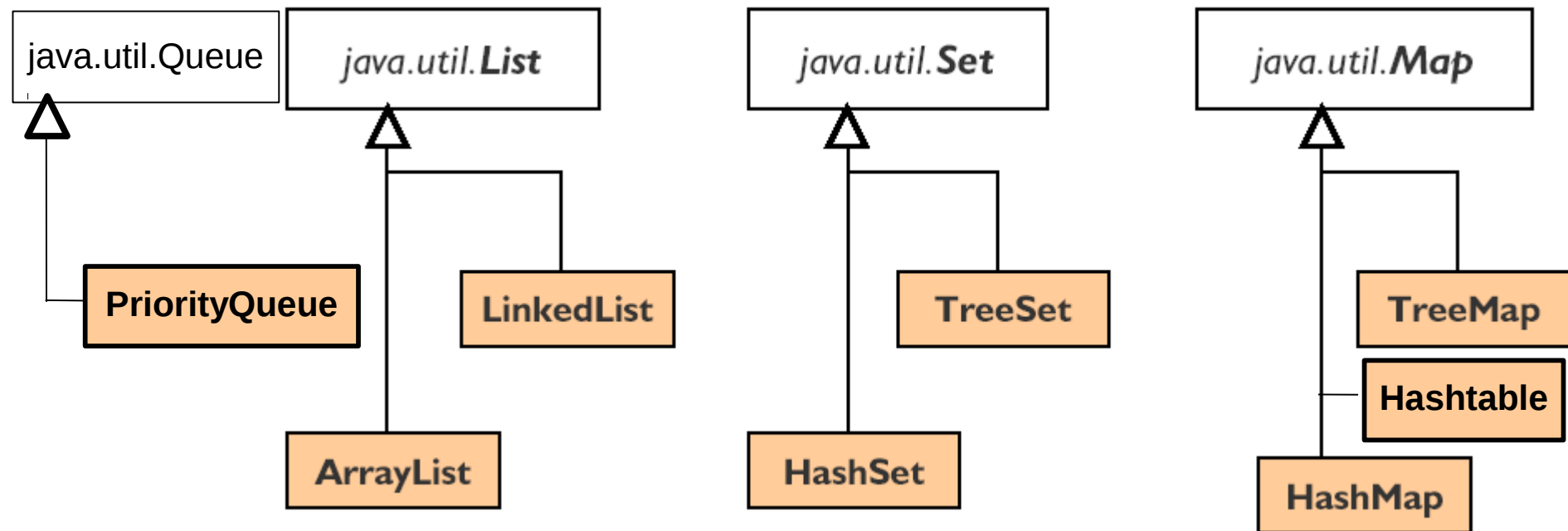
- *seqüência arbitrária*
- *elementos não repetem*

*Coleções de
pares de elementos*

*java.util.**Map***

- *Pares chave/valor*

Implementações da estrutura de coleções



Interface **Collection**

- Operações básicas:
 - adiciona elemento: **add (Object o)**
 - remove elemento: **remove (Object o)**
- Operações de volume:
 - adiciona coleção: **addAll (Collection c)**
 - remove coleção: **removeAll (Collection c)**
 - mantém coleção: **retainAll (Collection c)**
 - remove todos os elementos: **clear()**
- retorna um objeto **Iterator** para percorrer a coleção: **iterator()**
- **int size()**
- **boolean isEmpty()**
- **boolean contains (Object o)**

Interface **List**

- Fornece adicionalmente métodos para:
 - manipular elementos via seus índices. Ex:
 - **add(int index, Object o)**: Adiciona elemento. O tamanho da lista aumenta em 1.
 - **remove(int index)**: Remove elemento da posição especificada e move todos os elementos após o elemento removido diminuindo o tamanho da lista em 1.
 - **set(int index, Object o)**: Substitui elemento. O tamanho da lista permanece igual.
 - manipular um intervalo específico de elementos. Ex:
 - **addAll(int index, Collection c)** : Insere na posição especificada
 - **subList(int fromIndex, int toIndex)**: obtém uma parte da lista, o índice final não faz parte do intervalo. Qualquer alteração na sublista também será feita na lista original (view)
 - recuperar elementos
 - **get(int index)**
 - retorna um objeto **ListIterator** para percorrer a lista:
listIterator()

Interface Iterator

Essa interface permite ver qualquer coleção como uma estrutura sequencial

- Determinar se a coleção tem mais elementos: `hasNext()`
- Obter uma referência ao próximo elemento da coleção: `next()`
- Apagar o último item retornado pelo método `next()`: `remove()`

Interface ListIterator

- Fornece adicionalmente os seguintes métodos:
 - determinar se há mais elementos ao percorrer a lista em ordem invertida: `hasPrevious()`
 - Obter uma referência ao elemento anterior da lista: `previous()`
 - para substituir o último item retornado pelo método `next()` ou `previous()`: `set(Object o)`
 - adiciona um objeto na posição atualmente apontada pelo iterador: `add(Object o)`

Classe **Collections**

- A classe **Collections** fornece métodos **static** que manipulam as coleções.
- Esses métodos implementam algoritmos para:
 - busca
 - ordenação
 - menor elemento da coleção
 - maior elemento da coleção, etc.

Algoritmos de coleções

– Exemplos de algoritmos que operam em objetos do tipo **List**:

- **sort**: classifica os elementos da lista

```
Collections.sort( list);
```

- **binarySearch**: localiza um elemento da lista

```
int result=Collections.binarySearch(list,key);
```

- **reverse**: inverte os elementos da lista

```
Collections.reverse( list);
```

- **shuffle**: "embaralha" os elementos da lista

```
Collections.shuffle(list );
```

Algoritmos de coleções

– Exemplos de algoritmos que operam em objetos do tipo **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.

```
int result=Collections.frequency(list,key);
```

- **disjoint**: determina se duas coleções não têm nenhum elemento em comum.

```
boolean result=Collections.disjoint(list1,list2);
```

Algoritmos de coleções

- Exemplos de algoritmos que operam em objetos do tipo **Collection**:
 - **reverseOrder**: retorna um comparador com a ordem natural invertida de uma coleção que implementa a interface Comparable.

Algoritmo Sort

- O algoritmo **sort** classifica (ordena) os elementos de uma lista **List**
- A ordem entre os elementos da lista é determinada pela **ordem natural** do tipo dos elementos
- Podemos especificar o segundo argumento do método **sort**, que é um objeto **Comparator**, para determinar uma ordem alternativa dos elementos

Algoritmo Sort

O algoritmo **sort** usa uma otimização do algoritmo merge sort que é rápida e estável:

- Rápida: é garantido que roda em tempo $n \log(n)$.
- Estável: mantém a ordem dos elementos iguais.

Importante quando desejamos ordenar a mesma lista várias vezes com diferentes atributos.

Algoritmo `binarySearch`

- O algoritmo `binarySearch` usa a busca binária que é $O(\log n)$. O algoritmo supõe que a lista está ordenada de forma crescente pela *ordem natural* do tipo dos elementos.
- Se a lista não está ordenada de forma crescente o resultado é indeterminado.
- Podemos especificar o segundo argumento do método `binarySearch`, que é um objeto `Comparator`. O método supõe que a lista está ordenada em forma crescente na ordem alternativa especificada pelo `Comparador`.

Algoritmo `binarySearch`

- O método `binarySearch` devolve o índice em que a chave é encontrada. Se a chave não está na lista, o método devolve $-(\text{insertion point}) - 1$.

Algoritmo max e min

- Os métodos max e min retornam o maior e menor elemento em uma coleção respectivamente de acordo com a ordem natural dos elementos.
- Podemos especificar um segundo argumento que é um objeto **Comparator**. O método retorna o máximo (mínimo) de acordo com o Comparador especificado.

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 = new LinkedList< String >(Arrays.asList( suits));

        // 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);
        Collections.binarySearch( list,"Hearts");
    } // fim do método printElements
```

Exemplo

```
import java.util.*;
import java.util.List;
import java.util.ArrayList;

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

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

        // 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);
        Collections.binarySearch( list,"Hearts");
    } // fim do método printElements
}
```

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

Exemplo

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

public class Sort1 {
    private static final List<String> list =
        Collections.singletonList("Hearts", "Spades", "Clubs", "Diamonds");

    public void printElements() {
        List<String> suits =
            Arrays.asList("Hearts", "Spades", "Clubs", "Diamonds");

        // 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);
        Collections.binarySearch( list, "Hearts");
    } // fim do método printElements
}
```

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

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" };

    private void printElements() {
        // imprime elementos do array
        ArrayList<String> list = new ArrayList<String>(Arrays.asList(suits));

        // classifica a lista
        Collections.sort(list); // classifica ArrayList

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

        Collections.binarySearch( list, "Hearts");
    } // fim do método printElements
}
```

Classifica a lista **list**
em ordem crescente

Exemplo

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

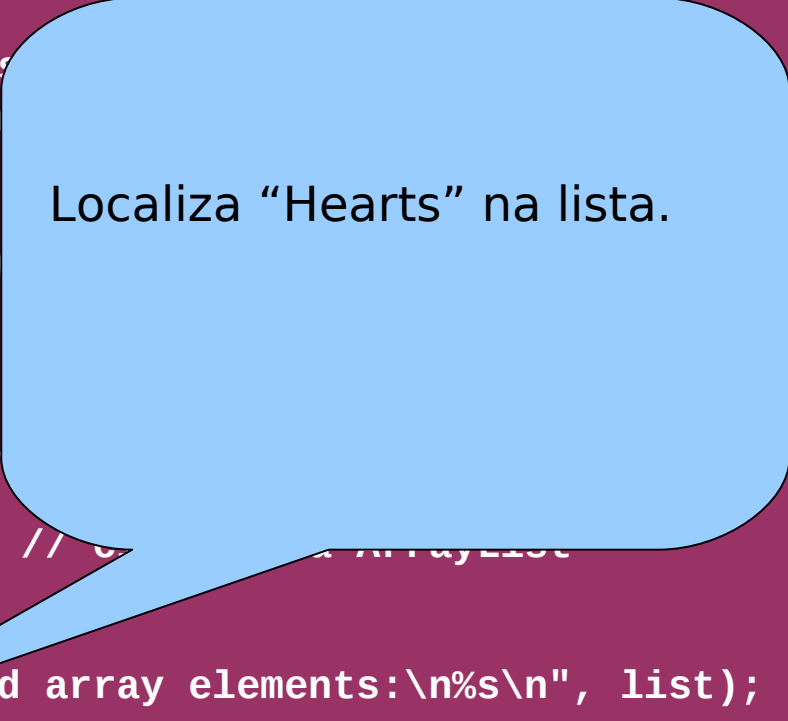
public class Sort1 {
    private static final String suits =
        { "Hearts", "Diamonds", "Clubs", "Spades" };

    public void printElements(){
        List< String > list = new ArrayList<>();
        >(Arrays.asList( suits));

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

        Collections.sort( list ); // Ordena a Array List

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



Localiza "Hearts" na lista.

Exemplo ordenar em ordem decrescente

```
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() {
        List< String > list = new ArrayList< String >
            >(Arrays.asList( suits));

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

        Collections.sort( list, Collections.reverseOrder());

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

reverseOrder:

retorna um comparador
com a ordem natural
invertida

Exemplo ordenar em ordem decrescente

```
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() {
        List< String > list = new ArrayList< String >
            >(Arrays.asList( suits));

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

        Collections.sort( list, Collections.reverseOrder());

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

A lista não está ordenada de forma crescente. Assim, o resultado é Indeterminado.

Exemplo Shuffle

```
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() {
        List< String > list = new ArrayList< String >
        >(Arrays.asList( suits));

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

        Collections.shuffle( list);

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

“embaralha” os elementos da lista.

Shuffle é útil por exemplo para embaralhar uma lista de cartas, para gerar casos de teste.

Exemplo Shuffle

```
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() {
        List< String > list = new ArrayList< String >
        >(Arrays.asList( suits));

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

        Collections.shuffle( list);

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

O resultado é
Indeterminado.

Exercícios:

- Se o elemento não está na lista, colocá-lo na posição correta na lista ordenada.
- Classificar os elementos de uma lista de `<Student>` em ordem crescente pelo número usp. Depois, buscar o aluno com número usp 4812035.

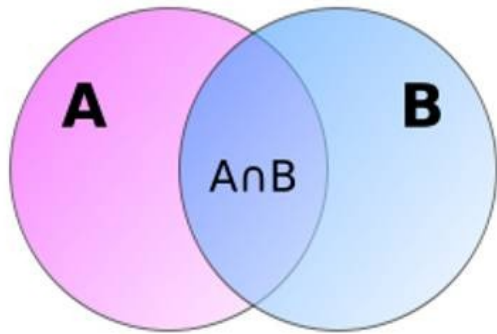
Comparator

```
public class StudentComparator implements Comparator <Student>{  
    public int compare(Student o1, Student o2) {  
        int returnValue;  
        if(o1.getId()==o2.getId())  
            returnValue=0;  
        else  
            if(o1.getId()>o2.getId())  
                returnValue=1;  
            else  
                returnValue=-1;  
        return returnValue;  
    }  
}
```

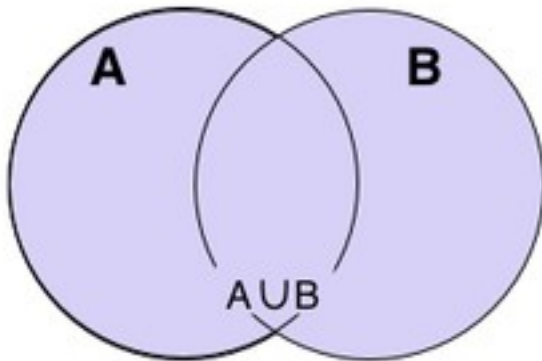
Conjuntos

- Um conjunto (**Set**) é uma coleção que contém elementos únicos não duplicados
- A ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.

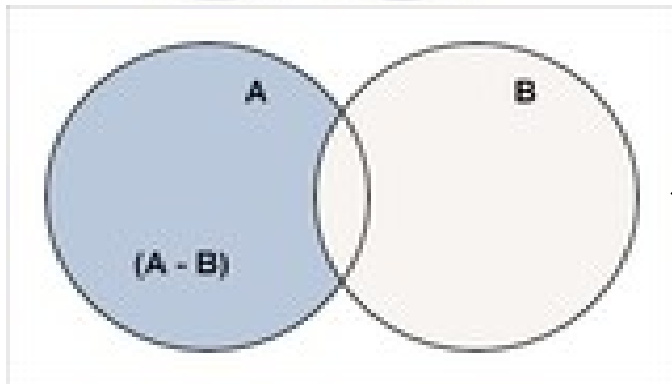
Conjuntos



retainAll



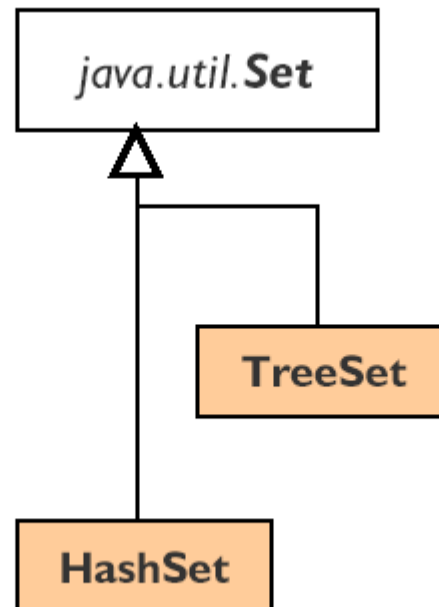
addAll



removeAll

Conjuntos

- As classe **HashSet** e **TreeSet** implementa a interface **Set**



HashSet – Exemplo 1

```
Set<String> conjunto = new HashSet<String>();  
conjunto.add("vermelho");  
conjunto.add("branco");  
conjunto.add("amarelo");  
conjunto.add("vermelho");  
System.out.println(conjunto);
```

– Saída do programa:

```
[branco, amarelo, vermelho]
```


HashSet – Exemplo 1

```
Set<String> conjunto = new HashSet<String>();  
conjunto.add("vermelho");  
conjunto.add("branco");  
conjunto.add("amarelo");  
conjunto.add("vermelho");  
System.out.println(conjunto.contains("amarelo"));
```

– Saída do prog

true

utiliza **tabelas de espalhamento** (hash tables), realizando a busca em tempo linear ($O(1)$), add e remove também são realizadas em $O(1)$.

HashSet e Tabelas de espalhamento

- Cada objeto é “classificado” pelo seu **hashCode**
- Agrupamos objetos pelo **hashCode**.
- Para buscar um objeto, procuramos somente entre os objetos que estão no grupo daquele **hashCode**.

HashSet – Exemplo 2

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 > mySet = new HashSet<String>(collection);  
    System.out.println("\n A coleção sem duplicatas: ");  
    System.out.println(mySet);  
} // 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 > mySet = new HashSet<String>(collection);  
    System.out.println("\n A coleção sem duplicatas: ");  
    System.out.println(mySet);  
} // 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[] ){
        SetTest myTest=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
```

TreeSet e a Interface SortedSet

- 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**

TreeSet e a Interface SortedSet

- A ordem é definida pelo método de comparação entre seus elementos.
- Esse método é definido pela interface `java.lang.Comparable`.
- Ou, pode se passar um **Comparator** para seu construtor.

Classe TreeSet

- Alguns métodos:
 - **headSet**(Object e): obter um subconjunto (uma “view”) em que cada elemento é menor do que o Elemento e
 - **tailSet**(Object e): obter um subconjunto (uma “view”) em que cada elemento é maior ou igual do que o Elemento e
 - **first()**: obter o primeiro elemento do conjunto
 - **last()**: obter o último elemento do conjunto

TreeSet - Exemplo 1

```
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;
public class SortedSetTest
{
    private static final String names[] = { "amarelo", "verde", "preto",
"marrom", "cinza", "branco", "laranja", "vermelho", "verde" };
    // create a sorted set with TreeSet, then manipulate it
    public SortedSetTest()
    {
        SortedSet<String> tree = new TreeSet<String>( Arrays.asList( names )
);
        System.out.println( "conjunto ordenado: " );
        System.out.println( tree );
    }
    public static void main( String args[] ){
        SortedSetTest myTest=new SortedSetTest();
    } // end main
} // end class SortedSetTest
```

TreeSet - Exemplo 1

```
import java.util.Arrays;
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetTest {

    private static final String names[] = { "amarelo", "preto",
    "marrom", "cinza", "branco", "laranja", "vermelho" };

    // create a sorted set with TreeSet, then manipulate it
    public SortedSetTest()
    {
        SortedSet<String> tree = new TreeSet<String>( Arrays.asList( names )
    );

        System.out.println( "conjunto ordenado: " );
        System.out.println( tree );
    }

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

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

TreeSet - Exemplo 1

- Saída do programa:

conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

TreeSet-Exemplo 1

Chama o método **headSet** da classe **TreeSet** para obter um subconjunto (uma “view”) em que cada elemento é menor do que “laranja”

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

Saída:

conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

headSet("laranja"): amarelo branco cinza

TreeSet-Exemplo 1

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

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

Saída:

conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

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

TreeSet-Exemplo 1

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

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

Saída:

conjunto ordenado:

amarelo branco cinza laranja marrom preto verde vermelho

first: amarelo

last: vermelho

TreeSet x HashSet

- Um TreeSet utiliza uma árvore rubro-negra como implementação
- Um HashSet usa uma tabela de espalhamento como implementação
- Um TreeSet gasta computacionalmente $O(\log(n))$ para inserir, enquanto o HashSet gasta apenas $O(1)$.

List x Set

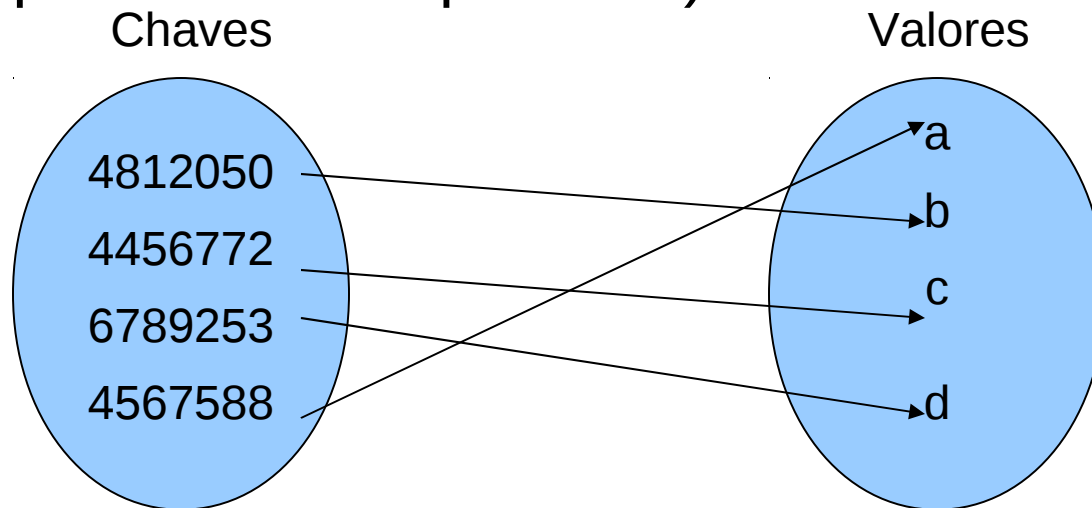
- A interface **List** permite elementos duplicados, enquanto **Set** define um conjunto de elementos únicos.
- **List** mantem a ordem em que os elementos foram adicionados.
- A busca em um **Set** pode ser mais rápida do que em um objeto do tipo **List**



- Dada alguma informação sobre um objeto, gostaríamos de buscá-lo rapidamente
- Como fazer isso?

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)



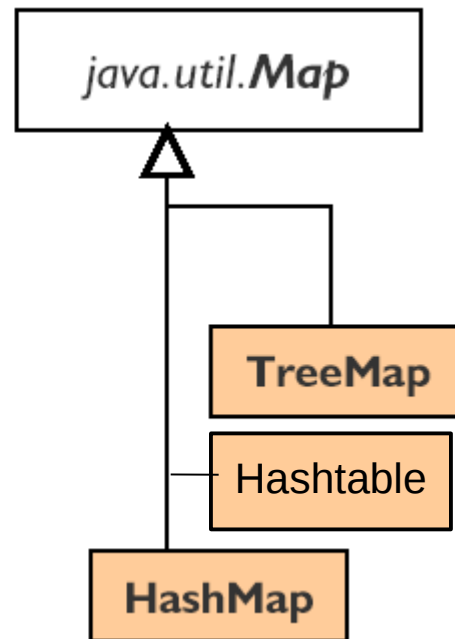
Interface Map

- Operações básicas:
 - adiciona par: **put (Object key, Object value)**
 - devolve valor associado à chave: **get(Object key)**
 - remove par: **remove (Object key)**
- Operações de volume:
 - adiciona mapa: **putAll (Map m)**
 - obtém o conjunto de chaves: **keySet()**
 - obtém a coleção de valores: **values()**
 - remove todos os pares: **clear()**
- **int size()**
- **boolean isEmpty()**
- **boolean containsKey (Object key)**
- **boolean containsValue (Object value)**

Mapas

Classes que implementam a interface **Map**

- **HashMap**
- **TreeMap**
- **Hashtable**



HashMap – Exemplo 1

```
Map<Integer,String> mapa=new HashMap<Integer, String>();  
mapa.put(455,"vermelho");  
mapa.put(333,"branco");  
mapa.put(678,"amarelo");  
mapa.put(455,"azul");  
System.out.println(mapa);
```

– Saída do programa:

```
{455=azul, 678=amarelo, 333=branco}
```

HashMap – Exemplo 1

```
Map<Integer, String > mapa = new Hashmap<Integer, String>();  
mapa.put(455,"vermelho");  
mapa.put(333,"branco");  
mapa.put(678,"amarelo");  
mapa.put(455,"azul");  
System.out.println(mapa.keySet());  
System.out.println(mapa.values());
```

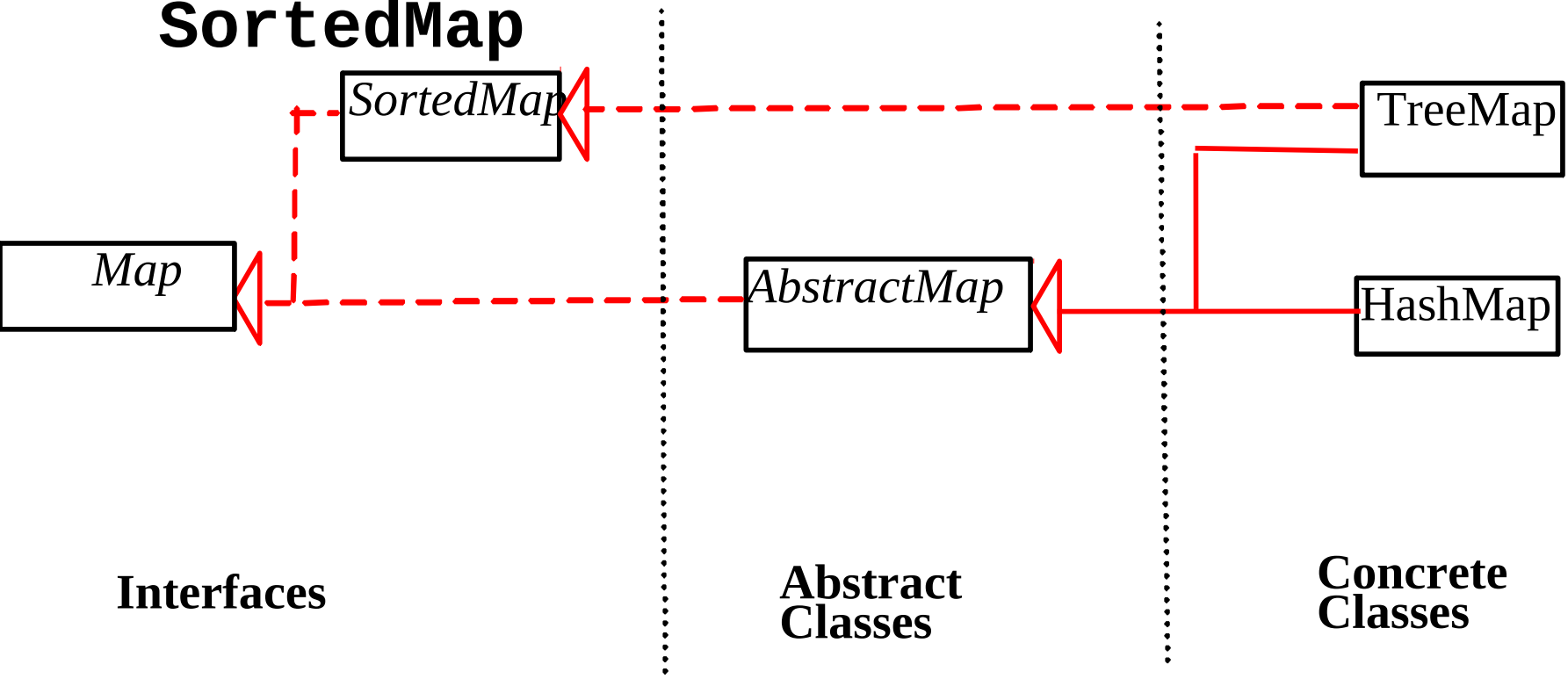
– Saída do programa:

```
[455, 678, 333]
```

```
[azul, amarelo, branco]
```

Mapas ordenados

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



Classe TreeMap

- Alguns métodos:
 - **headMap**(Object key): obter um subconjunto (uma “view”) do mapa em que cada par tem chave menor do que key
 - **tailMap**(Object key): obter um subconjunto (uma “view”) do mapa em que cada par tem chave maior ou igual do que key
 - **firstKey**(): obter a primeira chave do mapa
 - **lastKey**(): obter a última chave do mapa

TreeMap – Exemplo 1

```
Map<Integer, String > mapa = new TreeMap<Integer, String>();  
mapa.put(455,"vermelho");  
mapa.put(333,"branco");  
mapa.put(678,"amarelo");  
mapa.put(455,"azul");  
System.out.println(mapa);
```

– Saída do programa:

```
{333=branco, 455=azul, 678=amarelo}
```

TreeMap – Exemplo 1

```
Map<Integer, String > mapa = new TreeMap<Integer, String>();  
mapa.put(455,"vermelho");  
mapa.put(333,"branco");  
mapa.put(678,"amarelo");  
mapa.put(455,"azul");  
System.out.println(mapa);  
System.out.println(mapa.firstKey());  
System.out.println(mapa.lastKey());  
System.out.println(mapa.headMap(455));  
System.out.println(mapa.tailMap(455));
```

```
{333=branco, 455=azul, 678=amarelo}  
333  
678  
{333=branco}  
{455=azul, 678=amarelo}
```

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método de comparação entre seus elementos.
- Opção 1:
 - Fazer com que a classe dos elementos se torne “comparável” implementando a interface **Comparable** e incluindo o método **compareTo**

TreeSet e TreeMap

- A ordem nessa estrutura é dada pelo método `compareTo` dos elementos.
- Opção 1:
 - Fazer com que a classe dos elementos torne “comparável” implementando a interface `Comparable` e incluindo o método **`compareTo`**

Este método deve retornar **zero**, se o objeto comparado for igual a este objeto, um número **negativo**, se este objeto for menor que o objeto dado, e um número **positivo**, se este objeto for maior que o objeto dado.

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método de comparação entre seus elementos.
- Opção 1:
 - Fazer com que a classe dos elementos se torne “comparável” implementando a interface **Comparable** e incluindo o método **compareTo**
- Opção 2:
 - Criar uma classe que implementa a interface **Comparator**, e incluir o método **compare**.

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método de comparação entre seus elementos.
- Opção 1:
 - Fazer com que a classe seja “comparável” implementando a interface `Comparable` e incluindo o método `compareTo`.
- Opção 2:
 - Criar uma classe que implementa a interface **Comparator**, e incluir o método **compare**.

Esse método compara dois objetos e retorna um inteiro **negativo** se o primeiro for menor do que o segundo; **zero**, se forem idênticos; e um valor **positivo**, caso contrário.

Comparable

```
public class Employee implements Comparable{
    private int id;
    private String name;
    public Employee(int i, String n){
        this.id=i;
        this.name=n;
    }
    public int getId(){
        return id;
    }
    public String getName(){
        return name;
    }
    public int compareTo(Object obj){
        int returnValue;
        if(this.id==((Employee) obj).getId())
            returnValue=0;
        else
            if(this.id>((Employee) obj).getId())
                returnValue=1;
            else
                returnValue=-1;
        return returnValue;
    }
}
```


Comparator

```
public class Employee1{
    private int id;
    private String name;
    public Employee1(int i, String n){
        this.id=i;
        this.name=n;
    }
    public int getId(){
        return id;
    }
    public String getName(){
        return name;
    }
}
```

Comparator

```
public class EmployeeComparator implements Comparator
<Employee1>{

    public int compare(Employee1 o1, Employee1 o2) {
        int returnValue;
        if(o1.getId()==o2.getId())
            returnValue=0;
        else
            if(o1.getId(>o2.getId())
                returnValue=1;
            else
                returnValue=-1;
        return returnValue;
    }
}
```

HashSet e HashMap

- Usam tabela hash
- Para adicionar um objeto a uma tabela hash é calculado o **hashCode** do objeto.
- Para que isso funcione corretamente é necessário verificar que o método **hashCode** de cada objeto retorne o mesmo valor para dois objetos, se eles são considerados iguais:

Se **a.equals(b)** implica **a.hashCode() == b.hashCode()**

HashSet e HashMap

Para a classe Employee com:

```
private int id;  
private String name;
```

Incluimos os métodos `hashCode` e `equals`

```
public int hashCode() {  
    final int PRIME = 31;  
    int result = 1;  
    result = PRIME * result + id;  
    result = PRIME * result + ((name == null) ? 0 : name.hashCode());  
    return result;  
}
```

HashSet e HashMap

Para a classe Employee com:

```
private int id;  
private String name;
```

Podemos usar o Eclipse para isso: Ir no menu **Source** e depois a opção **Generate hashCode() and equals()**.

Incluimos os métodos **hashCode** e **equals**

```
public int hashCode() {  
    final int PRIME = 31;  
    int result = 1;  
    result = PRIME * result + id;  
    result = PRIME * result + ((name == null) ? 0 : name.hashCode());  
    return result;  
}
```

HashSet e HashMap

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    final Employee other = (Employee) obj;  
    if (id != other.id)  
        return false;  
    if (name == null) {  
        if (other.name != null)  
            return false;  
    } else if (!name.equals(other.name))  
        return false;  
    return true; }  
}
```

Exercício

Entre com uma frase:

To be or not to be: that is the question

Saída:

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

Classe StringTokenizer

Essa classe permite dividir a string de entrada em tokens. Um **token** é uma parte de um string.

Ex:

```
String s = "Five Three Nine One"
```

```
StringTokenizer st = new StringTokenizer(s);
```

Métodos:

- Determinar se a string tem mais tokens:
hasMoreTokens()
- Obter uma referência ao próximo token da string:
nextToken()
- Determinar o número de tokens na string:
countTokens()