



Coleções Java



Tópicos

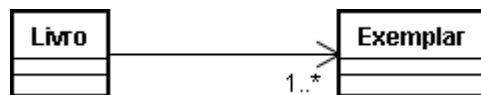
- utilizar arrays, lists, sets ou maps dependendo da necessidade do programa;
- iterar e ordenar listas e coleções;
- usar mapas para inserção e busca de objetos.

O que é uma Coleção?

O que é e por que o framework
Collection?

O que é uma coleção?

- Um objeto do tipo “coleção” – algumas vezes chamado de um recipiente (*container*) – é simplesmente um objeto que agrupa múltiplos elementos em uma única estrutura
- Coleções são usadas para armazenar, recuperar e manipular dados agregados
 - Tipicamente, elas são usadas para armazenar instâncias de objetos relacionados – exemplares de um livro, sócios de uma biblioteca, empréstimos de um exemplar, etc.
 - A necessidade de uso de uma coleção vem de associações com multiplicidade n no diagrama conceitual.



```
public class Livro {
    private Exemplar[] exemplares;
}
```



O que é o framework de Coleções?

- Um framework de coleções é uma arquitetura unificada para representar e manipular coleções.
- Todas as coleções pertencentes ao framework contém:
 - Interfaces
 - Implementações
 - Algoritmos



Benefícios de um framework de Coleções

- Quando comparados aos arrays Java, as Coleções têm uma série de vantagens:
 - Não podemos redimensionar um array em tempo de execução.
 - Não conseguimos saber quantas posições do array foram utilizadas sem para isso criar métodos auxiliares.
 - É complicado remover elementos do meio do array.
- O uso de interfaces no framework Collection possibilita o uso polimórfico das implementações dessas interfaces.
- Reduz o esforço de programação uma vez que várias dessas implementações estão disponíveis.
- Estão disponíveis ainda uma série de algoritmos que operam sobre as implementações de Collection (ordenações, buscas, etc.)



Interfaces e Implementações do framework Collection



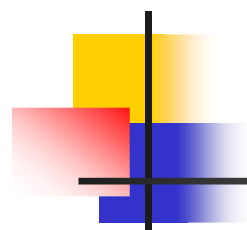
Interfaces

- As interfaces do framework Collection são tipos abstratos de dados que representam coleções.
 - Estas interfaces se apresentam na forma de interfaces Java
- As interfaces permitem tratar as coleções independentemente dos detalhes de implementação da sua representação
 - Comportamento polimórfico
- Na linguagem de programação Java (e em outras linguagens orientadas a objetos), as interfaces geralmente são organizadas na forma de uma hierarquia.
 - Dentre essas, você escolhe aquela que fornece um tipo que atende suas necessidades



Implementações

- São as implementações concretas das interfaces do framework Collection



Implementações de propósito geral

Implementações de propósito geral

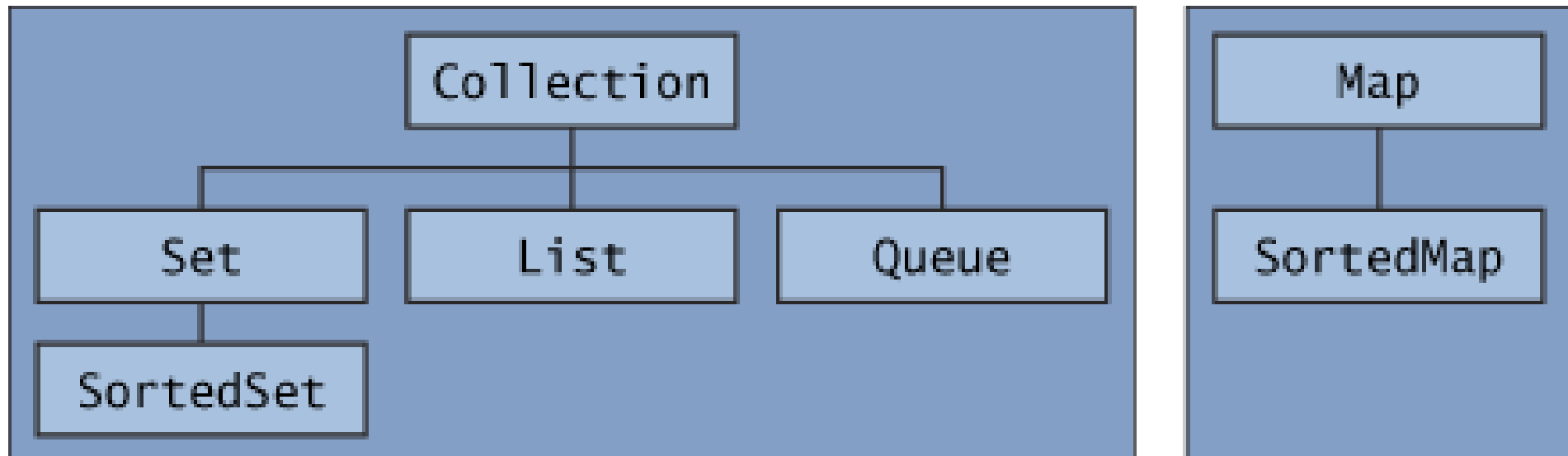
Interfaces	Implementações				
	Tabelas Hash	Arrays de tamanho variável	Árvores	Listas encadeadas	Tabelas Hash + Listas encadeadas
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap



As interfaces principais do framework Collection



A hierarquia das interfaces principais





As interfaces principais do framework Collection

- As interfaces principais são a base do framework Java Collection
- Essas interfaces são relacionadas por uma hierarquia de herança
 - Você pode criar uma nova interface Collection a partir delas, mas é altamente improvável que você tenha de fazer isso.



A interface Collection



A interface Collection

- A raiz da hierarquia de interfaces
- Representa o conjunto mínimo de operações que toda interface Collection implementa
 - Todo objeto que representa uma coleção é um tipo de interface Collection
- É utilizada para passar como parâmetros objetos do tipo Collection quando o máximo de generalidade é necessário
 - Use a interface Collection como um tipo
- O JDK não fornece nenhuma implementação direta desta interface, mas fornece implementações das sub interfaces mais específicas, tais como Set e List



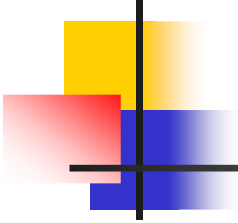
A interface Collection

```
public interface Collection<E>
    extends Iterable<E> {
    // Query Operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);

    // Modification Operations
    boolean add(E o);
    boolean remove(Object o);

    // Bulk Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
}
```

<<interface>> Collection
<div data-bbox="1989 288 2033 336">E</div> <div>+ size() : int + isEmpty() : boolean + contains(o : Object) : boolean + iterator() : Iterator<E> + add(e : E) : boolean + remove(o : Object) : boolean + clear() : void</div>



Exemplo: Uso da interface Collection como um tipo

```
class Livro {
    // qualquer implementação de Collection
    // pode ser usada
    private HashSet exemplares = new HashSet();

    public Collection getExemplares() {
        return exemplares;
    }
}

public class Main {
    public static void main(String[] args) {
        // obtem uma instancia de livro...
        Collection exemplares = livro.getExemplares();
    }
}
```



A interface Collection

As operações `add()` e `remove()`



Os métodos add() e remove() da interface Collection

- O método add() é definido genericamente o bastante de modo que algumas implementações aceitam elementos duplicados e outras não.
- Ele garante que, após a chamada do método, a coleção conterá o elemento especificado. O método retorna true se a coleção mudou em função da chamada.
 - O método add() na interface Set segue a regra das “não duplicatas”



A interface Collection

Percorrendo a coleção



Duas formas de percorrer coleções

- for-each

- A construção for-each permite de forma concisa percorrer um array ou uma coleção usando um laço for

```
for (Object o: collection)  
    System.out.println(o);
```

- Iterator

- Um Iterator é um objeto que permite percorrer uma coleção e, se necessário, remover elementos seletivamente.



Um exemplo de uso do for-each

```
class Livro {
    private HashSet exemplares = new HashSet();

    public Collection getExemplares() {
        return exemplares;
    }
}

public class Main {
    public static void main(String[] args) {
        // obtem uma instancia de livro...

        Collection exemplares = livro.getExemplares();
        for (Object o: exemplares) {
            System.out.println(((Exemplar)o).getCodigo());
        }
    }
}
```



Um exemplo de uso do for-each

- A partir da versão 5 do Java é possível fazer:

Generics

```
class Livro {  
    private HashSet<Exemplar> exemplares =  
        new HashSet<Exemplar>();  
  
    public Collection<Exemplar> getExemplares() {  
        return exemplares;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // obtem uma instancia de livro...  
  
        Collection<Exemplar> exemplares =  
            livro.getExemplares();  
        for (Exemplar e: exemplares) {  
            System.out.println(e.getCodigo());  
        }  
    }  
}
```



A interface Iterator

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

- O método `hasNext()` retorna `true` se a coleção tem mais elementos
- O método `next()` retorna o próximo elemento na coleção
- O método `remove()` é a única maneira segura de modificar uma coleção durante a iteração; se a coleção for modificada de qualquer outra forma durante a iteração, o resultado é imprevisível



Um exemplo de uso do iterator

```
class Livro {
    private HashSet<Exemplar> exemplares =
        new HashSet<Exemplar>();

    public Collection<Exemplar> getExemplares() {
        return exemplares;
    }
}

public class Main {
    public static void main(String[] args) {
        // obtem uma instancia de livro...

        Collection<Exemplar> exemplares =
            livro.getExemplares();
        Iterator<Exemplar> it = exemplares.iterator();
        while (it.hasNext()) {
            Exemplar e = it.next();
            System.out.println(e.getCodigo());
        }
    }
}
```



Use o Iterator ao invés do for-each quando for necessário:

- Remover elementos. O for-each esconde o iterator, de modo que você não pode invocar o método remove(). Dessa forma, o for-each não pode ser usado para, por exemplo, filtrar os elementos em uma coleção

```
class Livro {  
    private HashSet<Exemplar> exemplares =  
        new HashSet<Exemplar>();  
  
    public Collection<Exemplar> getExemplares() {  
        return exemplares;  
    }  
  
    public void filtra(String codigo) {  
        Iterator<Exemplar> it = exemplares.iterator();  
        while (it.hasNext()) {  
            Exemplar e = it.next();  
            if (e.getCodigo().equals(codigo)) {  
                it.remove();  
            }  
        }  
    }  
}
```



Use o Iterator ao invés do for-each quando for necessário:

- Percorrer múltiplas coleções em paralelo

```
while (it.hasNext() && it.hasNext()) {  
    Exemplar e1 = it.next();  
    Exemplar e2 = it.next();  
}
```



A interface Set e suas implementações

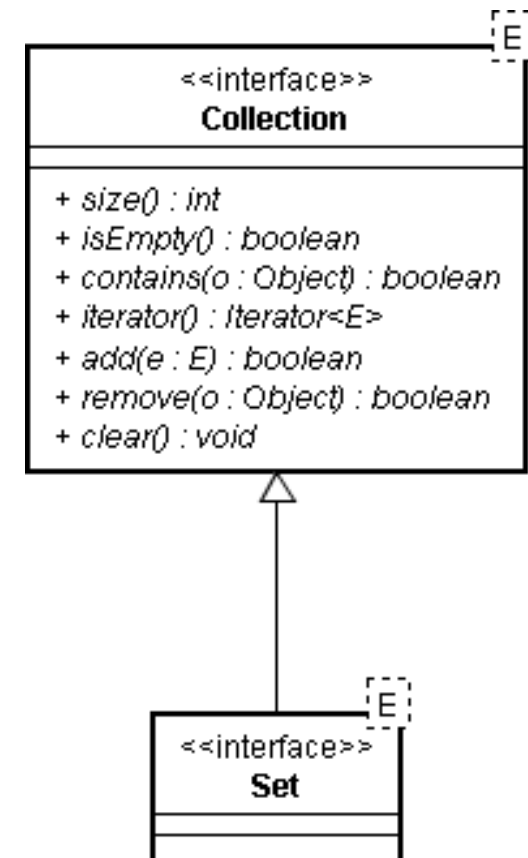


A interface Set

- Uma coleção **que não pode conter elementos duplicados**
- Modela a abstração matemática do conjunto e é usada para representar conjuntos
- A interface Set contém apenas os métodos herdados de *Collection*. Ela somente adiciona a restrição de que elementos duplicados são proibidos.

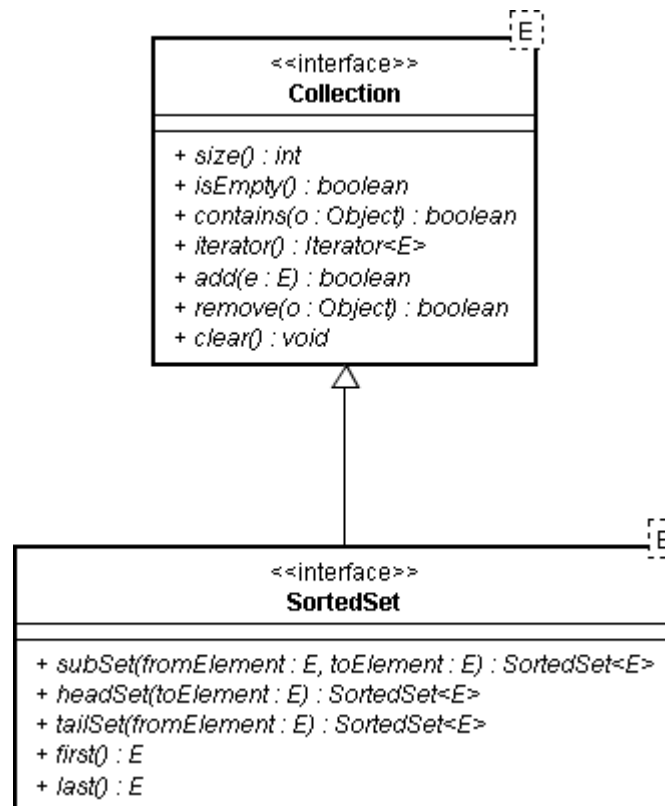
A interface Set (Java SE 5)

```
public interface Set<E> extends Collection<E> {  
    // Query Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object o);  
    Iterator<E> iterator();  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
  
    // Modification Operations  
    boolean add(E o);  
    boolean remove(Object o);  
  
    // Bulk Operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    boolean retainAll(Collection<?> c);  
    boolean removeAll(Collection<?> c);  
    void clear();  
}
```



A interface SortedSet

- Um Set que mantém seus elementos em ordem crescente. Várias operações adicionais são fornecidas para tirar proveito da ordenação



- SortedSets são usados para armazenar conjuntos naturalmente ordenados, tais como listas de nomes e registros de títulos



Implementações da Interface Set

- HashSet
- TreeSet
- LinkedHashSet



HashSet

- *HashSet* é muito mais rápido do que *TreeSet* (tempo constante versus tempo logarítmico para a maioria das operações) mas não oferece nenhuma garantia de ordenação.
- A implementação mais comumente usada



Exemplo: a interface Set & HashSet

```
class Verbete {
    private String verbete;
    private Set<String> definicoes = new HashSet<String>();

    public Verbete(String verbete) {
        this.verbete = verbete;
    }

    public void adicionaDefinicao(String definicao) {
        if (!definicoes.add(definicao)) {
            System.out.println("definição duplicada");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Verbete v = new Verbete("triângulo");
        v.adicionaDefinicao("tipo de estrangulamento");
        v.adicionaDefinicao("Um polígono com três lados.");
        v.adicionaDefinicao("tipo de estrangulamento");
    }
}
```



TreeSet

- Uma implementação da interface *SortedSet*.
- Usada quando você precisa usar os métodos na interface *SortedSet*, ou se é necessário percorrer a coleção *ordenada segundo seus valores*.




Exemplo: a interface Set & TreeSet

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

public class Main {
    public static void main(String[] args) {
        Set ts = new TreeSet();
        ts.add("um");
        ts.add("dois");
        ts.add("tres");
        ts.add("quatro");
        ts.add("tres");
        System.out.println("Membros do TreeSet = " + ts);
    }
}
```

Observe que o método
toString() na classe
TreeSet foi sobrescrito.



Saída:

Membros do TreeSet = [dois, quatro, tres, um]



LinkedHashSet

- Implementada como uma tabela hash suportada por uma lista encadeada
- Permite percorrer a coleção na ordem de inserção (da inserção mais antiga para a mais recente) e é quase tão rápida quanto o *HashSet*
- Livra os clientes da ordem caótica fornecida pelo *HashSet* sem incorrer no elevado custo associado ao *TreeSet*



Exemplo: a interface Set & LinkedHashSet

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

public class Main {
    public static void main(String[] args) {
        Set ts2 = new LinkedHashSet();
        ts2.add(2);
        ts2.add(1);
        ts2.add(3);
        ts2.add(3);
        System.out.println("Membros da LinkedHashSet = " + ts2);
    }
}
```

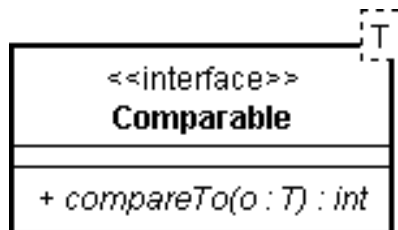
Saída:

Membros da LinkedHashSet = [2, 1, 3]



A ordem natural dos elementos

- Um *TreeSet* permite percorrer os elementos do conjunto segundo a “ordem natural dos elementos”.
- É fácil ordenar os elementos de um conjunto quando eles são números ou Strings, mas o que é a ordem natural de livros, por exemplo?
- Os elementos em um *TreeSet* têm de implementar a interface *Comparable*



- O método *compareTo()* retorna um número inteiro, negativo, nulo, ou positivo, no caso em que esse objeto seja menor, igual ou maior do que o objeto recebido como parâmetro



Exemplo

```
class Exemplar implements Comparable<Exemplar> {
    private int codigo;

    public int compareTo(Exemplar o) {
        return this.codigo - o.codigo;
    }

    // outros atributos e métodos
}

class Livro {
    private Set<Exemplar> exemplares =
        new TreeSet<Exemplar>();

    public void adicionaExemplar(Exemplar e) {
        exemplares.add(e);
    }

    // outros atributos e métodos
}
```




A interface List e suas implementações



A interface List

- Uma coleção ordenada (algumas vezes chamada de uma seqüência)
- Listas podem conter elementos duplicados
- O usuário de um List tem, geralmente, controle preciso sobre onde cada elemento é inserido e pode acessar os elementos pelo seu índice inteiro (sua posição)



Operações adicionais que a Interface List adiciona à Interface Collection

- Acesso posicional – manipula os elementos baseado na sua posição numérica na lista
- Busca – procura por uma objeto específico na lista e retorna sua posição numérica
- Percorrer a coleção – estende a semântica do Iterator para tirar proveito da natureza seqüencial da lista
- Subconjuntos – realiza operações em faixas de elementos da lista



A interface List

```
public interface List<E> extends Collection<E> {
    // Bulk Operations
    boolean addAll(int index, Collection<? extends E> c);

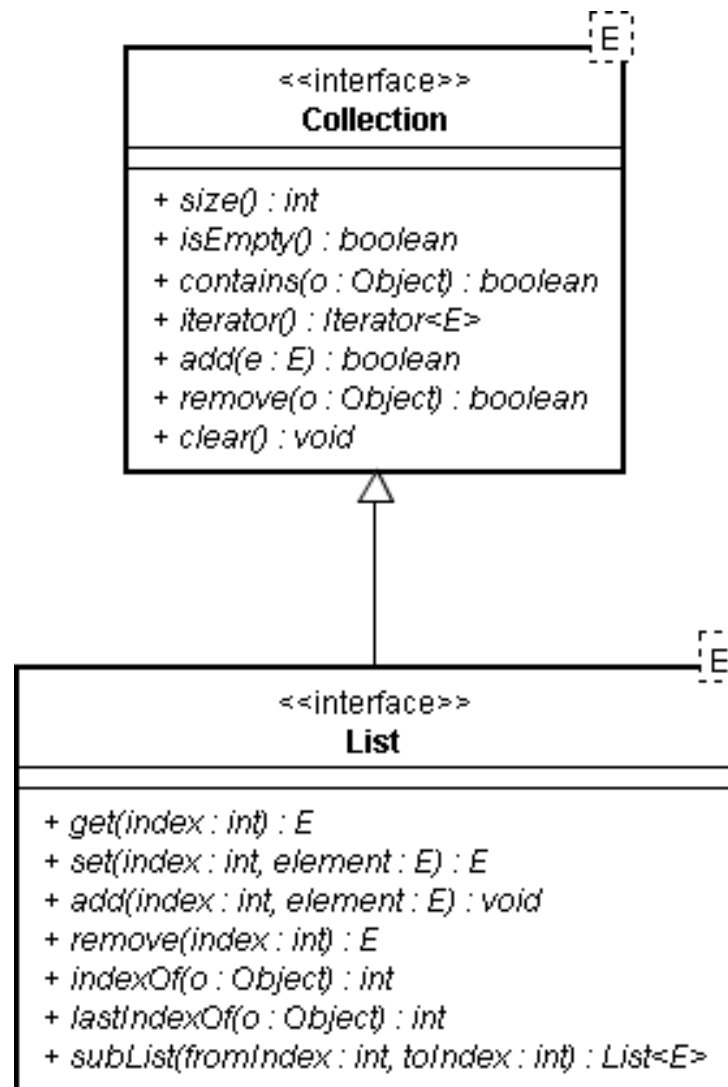
    // Positional Access Operations
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);

    // Search Operations
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // List Iterators
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // View
    List<E> subList(int fromIndex, int toIndex);
}
```

A interface List





Implementações da Interface List


- ArrayList
 - Oferece acesso posicional em tempo constante
 - Rápida
 - A implementação mais comumente usada
- LinkedList
 - Use-a se, freqüentemente, você tem de adicionar elementos no início da lista ou percorrer a lista para apagar elementos do seu interior



Exemplo

- Um ArrayList pode ser percorrido pelo índice da posição.

```
class Livro {  
    private List<Exemplar> exemplares =  
        new ArrayList<Exemplar>();  
  
    public void adicionaExemplar(Exemplar e)    {  
        exemplares.add(e);  
    }  
  
    public void imprime()    {  
        for (int i = 0; i < exemplares.size(); i++) {  
            System.out.println(exemplares.get(i).getCodigo());  
        }  
    }  
  
    // outros atributos e métodos  
}
```





A classe Collections



A classe *Collections*

- A classe *Collections* fornece uma série de algoritmos como métodos estáticos da classe.
- O primeiro argumento é a coleção sobre a qual a operação será realizada.
- A maioria dos algoritmos fornecidos opera em instâncias de *List*, mas alguns poucos operam em instâncias de *Collection*



Ordenação: Collections.sort()

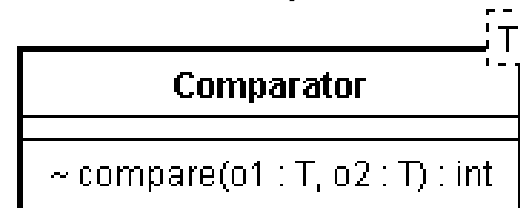
- Um dos algoritmos fornecidos pela classe *Collections* é o algoritmo de ordenação, implementado pelo método *sort()*. Este método ordena o conteúdo de um *List*.

```
class Livro {  
    private List<Exemplar> exemplares =  
        new ArrayList<Exemplar>();  
  
    public void adicionaExemplar(Exemplar e)    {  
        exemplares.add(e);  
        // ordena o conteúdo do List a cada inclusão  
        Collections.sort(exemplares);  
    }  
  
    public void imprime()    {  
        for (int i = 0; i < exemplares.size(); i++) {  
            System.out.println(exemplares.get(i).getCodigo());  
        }  
    }  
  
    // outros atributos e métodos  
}
```



Ordenação: Collections.sort()

- Para que possam ser ordenados, os elementos no *List* têm também de implementar a interface *Comparable*.
- Se você tentar ordenar um *List* cujos elementos não implementam a interface *Comparable*, o método *Collections.sort()* irá gerar uma *ClassCastException*.
- A “ordem natural” de comparação dos elementos – expressa pelo método *compareTo()* da interface *Comparable* – pode ser sobrescrita passando-se uma implementação de um *Comparator* a uma versão sobrecarregada do método *sort()*



- Exemplo: Vamos ordenar uma lista de Strings em ordem decrescente.



A interface Comparator

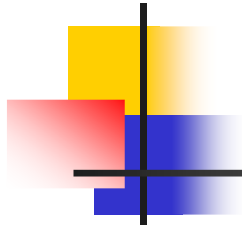
```
class Verbete {  
    // classe interna  
    class MeuComparator implements Comparator<String> {  
        public int compare(String s1, String s2) {  
            return -s1.compareTo(s2);  
        }  
    }  
  
    private MeuComparator mc = new MeuComparator();  
    private String verbete;  
    private List<String> definicoes = new ArrayList<String>();  
  
    public Verbete(String verbete) {  
        this.verbete = verbete;  
    }  
  
    public void adicionaDefinicao(String definicao) {  
        definicoes.add(definicao);  
        Collections.sort(definicoes, mc);  
    }  
}
```



Outros métodos úteis da classe Collections

Collections
<ul style="list-style-type: none"><u>+ sort(list : List<T>) : void</u><u>+ sort(list : List<T>, c : Comparator<T>) : void</u><u>+ binarySearch(list : List<Comparable<T>>, key : T) : int</u><u>+ reverse(list : List<?>) : void</u><u>+ shuffle(list : List<?>) : void</u><u>+ swap(list : List<?>, i : int, j : int) : void</u><u>+ fill(list : List<T>, obj : T) : void</u><u>+ copy(dest : List<T>, src : List<T>) : void</u><u>+ min(coll : Collection<T>) : T</u><u>+ max(coll : Collection<T>) : T</u><u>+ replaceAll(list : List<T>, oldVal : T, newVal : T) : boolean</u><u>+ addAll(c : Collection<T>, elements : T) : boolean</u>

- *E muitos outros...*



Embaralhando

- O algoritmo de embaralhamento (*shuffle*) faz o oposto do algoritmo de ordenação, isto é, ele destrói qualquer traço de ordem que podia haver no *List*.
 - Isto é, o algoritmo reordena o *List* de forma aleatória de modo que todas as permutações ocorram com igual probabilidade.
- Útil na implementação de jogos de azar
 - podem ser usados para embaralhar as cartas de um *List* representando um baralho
 - úteis para gerar conjuntos de testes



Manipulação rotineira de dados

- A classe *Collections* fornece cinco algoritmos para a manipulação rotineira de dados em objetos do tipo *List*
 - *reverse()* – reverte a ordem dos elementos em um *List*
 - *fill()* – sobrescreve cada elemento no *List* com o valor especificado. Esta operação é útil para reinicializar um *List*
 - *copy()* – toma dois argumentos, um *List* destino e um *List* origem e copia os elementos da origem no destino, sobrescrevendo seu conteúdo. O *List* de destino deve ser pelo menos do mesmo tamanho do *List* de origem. Se ele for maior, os elementos restantes no *List* de destino não são afetados.
 - *swap()* – permuta os elementos do *List* nas posições especificadas
 - *addAll()* – adiciona todos os elementos especificados à *Collection*. Os elementos a serem adicionados podem ser especificados individualmente ou como um array.



Busca

- A classe *Collections* tem o método *binarySearch()* para buscar um elemento específico em um *List* ordenado

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

public class Cliente {
    public static void main(String[] args) {
        // monta os dados do teste
        String[] nome = {"Sang", "Shin", "Boston",
                        "Passion", "Shin"};

        List<String> l = Arrays.asList(nome);
        Collections.sort(l);
        int posicao = Collections.binarySearch(l, "Sang");
        System.out.println("Posicao do item procurado = " + posicao);
    }
}
```




A Interface Map e suas Implementações



A interface Map

- Manipula pares chave/valor
- Um Map não pode conter chaves duplicadas; cada chave pode mapear no máximo um valor
 - Se você já usou uma Hashtable, você já está familiarizado com o básico da interface Map
- Maps são chamados de Dicionários em outras linguagens de programação

A Interface Map

```
public interface Map<K,V> {  
    // Query Operations  
    int size();  
    boolean isEmpty();  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    V get(Object key);  
  
    // Modification Operations  
    V put(K key, V value);  
    V remove(Object key);  
  
    // Bulk Operations  
    void putAll(Map<? extends K, ? extends V> t);  
    void clear();  
  
    // Views  
    Set<K> keySet();  
    Collection<V> values();  
    Set<Map.Entry<K, V>> entrySet();  
  
    interface Entry<K,V> {  
        K getKey();  
        V getValue();  
        V setValue(V value);  
    }  
}
```

Map
<div>K, V</div> ~ size() : int ~ isEmpty() : boolean ~ containsKey(key : Object) : boolean ~ containsValue(value : Object) : boolean ~ get(key : Object) : V ~ put(key : K, value : V) : V ~ remove(key : Object) : V ~ clear() : void ~ keySet() : Set<K> ~ values() : Collection<V>



A Interface SortedMap

- Um *SortedMap* mantém a coleção de pares chave/valor ordenada em ordem crescente, segundo a chave.
 - Este é o Map análogo ao SortedSet
- *SortedMaps* são usados para coleções de pares chave/valor naturalmente ordenadas, tais como dicionários e listas telefônicas

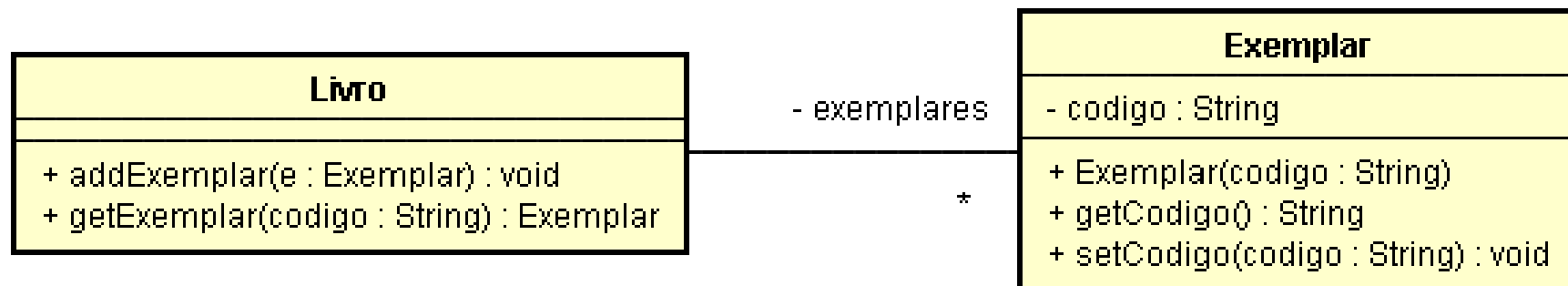


Implementações da Interface Map

- HashMap
 - Use-a quando você precisa de máxima velocidade e não se importa com a ordem da iteração sobre os elementos da coleção
 - A implementação mais comumente usada
- TreeMap
 - Use-a quando você precisa de operações da interface SortedMap ou percorrer a coleção na ordem da chave
- LinkedHashMap
 - Esta implementação oferece desempenho quase igual ao do HashMap e iteração na ordem de inserção



Maps versus Lists





Maps X Lists

A classe Exemplar

```
class Exemplar {
    private String codigo;

    public Exemplar(String codigo) {
        this.codigo = codigo;
    }

    public String getCodigo() {
        return codigo;
    }

    public void setCodigo(String codigo) {
        this.codigo = codigo;
    }
}
```



Maps X Lists

A classe Livro implementada com Lists

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

class Livro {
    private List<Exemplar> exemplares =
        new ArrayList<Exemplar>();

    public void addExemplar(Exemplar e) {
        exemplares.add(e);
    }

    public Exemplar getExemplar(String codigo) {
        for(Exemplar e: exemplares) {
            if(e.getCodigo().equals(codigo))
                return e;
        }
        return null;
    }
}
```




Maps X Lists

A classe Livro implementada com Maps

```
import java.util.HashMap;
import java.util.Map;

class Livro {
    private Map<String, Exemplar> exemplares =
        new HashMap<String, Exemplar>();

    public void addExemplar(Exemplar e) {
        exemplares.put(e.getCodigo(), e);
    }

    public Exemplar getExemplar(String codigo) {
        return exemplares.get(codigo);
    }
}
```