Modulo II – Tópicos em Java – Collections

Prde. Ismael H F Santos

Ementa

- Modulo II - Tópicos em JAVA
  - Collections

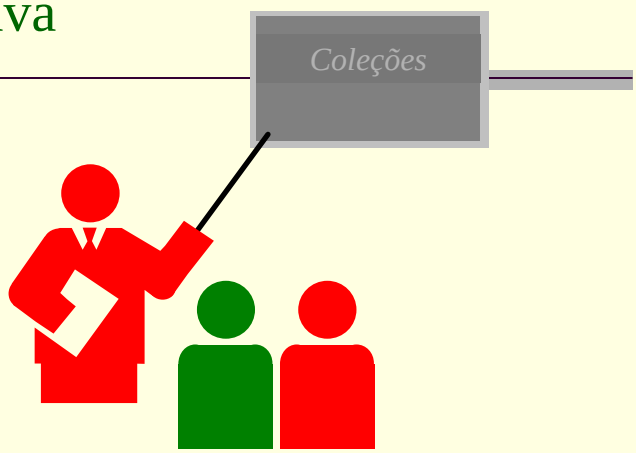
Bibliografia

- *Linguagem de Programação JAVA*
  - *Ismael H. F. Santos, Apostila UniverCidade, 2002*
- *The Java Tutorial: A practical guide for programmers*
  - *Tutorial on-line: http://java.sun.com/docs/books/tutorial*
- *Java in a Nutshell*
  - *David Flanagan, O´Reilly & Associates*
- *Just Java 2*
  - *Mark C. Chan, Steven W. Griffith e Anthony F. Iasi, Makron Books.*
- *Java 1.2*
  - *Laura Lemay & Rogers Cadenhead, Editora Campos*

Livros

- **Core Java 2**, Cay S. Horstmann, Gary Cornell
  - Volume 1 (Fundamentos)
  - Volume 2 (Características Avançadas)
- **Java: Como Programar**, Deitel & Deitel
- **Thinrei in Patterns with JAVA**, Bruce Eckel
  - **Gratuito**. http://www.mindview.net/Books/TIJ/

POO-Java



O que são coleções ?

- Coleções são estruturas de dados utilizadas para armazenar e manipular informações
  - objetos que representam um grupo de objetos
- A escolha de um tipo de estrutura depende dos requisitos do problema que se deseja resolver
- Versões anteriores a Java 2 deereciam
  - implementações de algumas estruturas de dados básicas: `Vector`, `Hashtable`, `Stack`, `Properties`, `BitSet` e a interface `Enumeration`

Collections Framework (CF)

- Arquitetura unificada para representar e manipular diferentes tipos de coleções. Exemplos famosos:
  - C++ Standard Template Library (STL)
  - Smalltalk's collection classes
- Essa arquitetura contem:
  - **Interfaces e Classes Abstratas (ex: List, AbstractList)**
    - **Tipos Abstratos de Dados que representam Collections**
  - **Implementações (ex: ArrayList)**
    - **Classes concretas que implementam os TADs**
  - **Algoritmos**
    - **Métodos como Pesquisa e Ordenação sobre classes concretas que implementam as interfaces do Collection Framework**

Benefícios do CF

- Redução do esforço de programação
- Maior eficiência e qualidade na programação
- Permite a interoperabilidade entre diferentes e não relacionadas APIs
- Reduz o esforço para aprender, usar e projetar novas APIs
- Reusabilidade

Interface

- Tipos abstratos de dados
  - permitem que coleções sejam manipuladas independentemente dos detalhes de sua implementação

```
public interface List {
  …
  boolean add(Object o);
  Object set (int index, Object o);
  Object get (int index);
  int indexde(Object o);
  Object remove(int index);
  …
}
```


Implementações

- Estruturas de dados reusáveis
  - implementações concretas das interfaces de coleções

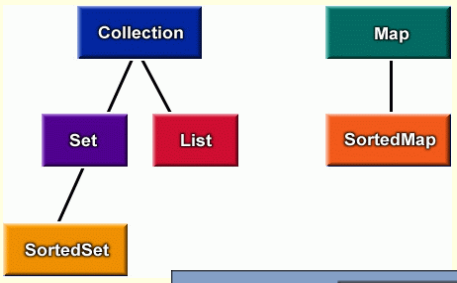
```
public class Vector extends AbstractList
  implements List, Cloneable, Serializable

public class LinkedList extends AbstractSequentialList
  implements List, Cloneable, Serializable
```

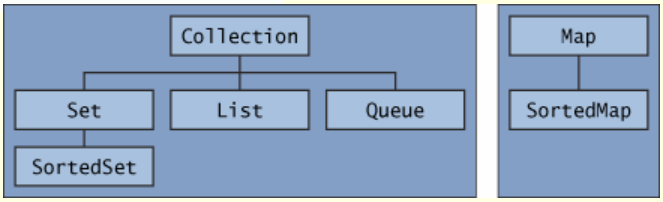
Algoritmos

- Métodos que executam computações úteis (como busca e ordenação) sobre objetos que implementam uma interface de coleção
- Os algoritmos são chamados de polimórficos porque o mesmo método pode ser utilizado sobre diferentes implementações de uma interface
  - Na essência temos o reuso de funcionalidade

Interfaces Básicas: hierarquia



©The Java™ Tutorial – java 1.2



©The Java™ Tutorial - java 6.0

Interface **Collection** – core CF

- Denominador comum de todas as coleções. Uma coleção representa um grupo de objetos chamados de elementos da coleção.
  - não há implementações “diretas” dessa interface
- Implementações dos tipos básicos de coleções tem construtores que recebem um parâmetro do tipo **Collection**
  - isso permite a criação de uma coleção contendo inicialmente todos os elementos da coleção especificada como parâmetro, independentemente de seu tipo ou implementação

Variações dos Tipos Básicos

- Para não explodir o numero de coleções básicas Java não provê interfaces separadas para variações dos tipos de coleção
  - mutável / imutável (read-only)
  - tamanho fixo/variável
  - append-only
- Todos os métodos que operam sobre uma coleção são considerados *opcionais*
  - uma implementação pode não suportar uma operação. Se uma operação não suportada é invocada será lançada a exceção UnsupportedOperationException
  - Cabe a implementação documentar quais operações suporta
  - As implementações Java implementam todas as operações opcioinais

Métodos de **Collection**

```
public interface Collection<E>  extends Iterable<E> {
  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);                          // Optional
  boolean remove(Object element);                  // Optional
  Iterator<E> iterator();
  // Bulk Operations                                  // Generic interface !
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c); // Optional
  boolean removeAll(Collection <?> c);          // Optional
  boolean retainAll(Collection <?> c);          // Optional
  void clear();                                    // Optional
  // Array Operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```


Collection Iterator - Iteradores

- Percorrer coleções, similares aos objetos **Enumeration**

```
public interface Iterator<E> {
  boolean hasNext();
  E next() throws NoSuchElementException;
  void remove() throws UnsupportedOperationException,
                       IllegalStateException;          // Optional !
}
```

- Remoção segura elementos da coleção sendo percorrida

```
List<String> list = new ArrayList<String>();
.....
static void filter(Collection<?> c) { // polimorfismo, funciona com qq objeto CF
  for (Iterator <?> i = c.iterator(); i.hasNext(); )
    if (!cond(i.next())) i.remove(); // remove da coleção elementos que não
}                                    // satisfazem uma dada condição
```

Collection

- **for-each loop**

```
Collection<String> c;
for (String s : collection)  System.out.println(s);
```

- **Bulk operations**
  - containsAll, addAll, removeAll, retainAll, clear

```
Collection<XX> c;   XX e;
c.removeAll(Collections.singleton(e));   // singleton static factory method !
```

- **Array operations**
  - toArray

```
Collection<XX> c;
Object[] a = c.toArray();            // retorna elementos no array de Object !
XX[] a = c. toArray(new XX[0]);
```


Interface **Set**

- Uma coleção que não pode conter elementos duplicados
  - corresponde à abstração de um **conjunto**
- Contem somente os métodos herdados de **Collection**. Java define tres implementações de proposito geral para Set: **HashSet**, **TreeSet** e **LinkedHashSet**
  - HashSet – armazena elementos numa hash table apresenta melhor performance mas não garante ordem na iteração
  - TreeSet– armazena elementos numa red-black tree e os ordena baseados nos seus valores. Performance bem inferior a HashSet
  - LinkedHashSet – armazena elementos numa hash table com uma linked list referenciando. Ordena os elementos pela ordem de inserção. Performance ligeiramente pior que HashSet
- Exemplos:
  - cursos no horário de um aluno
  - processos rodando em uma máquina

Métodos de **Set**

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


Implementação: **HashSet**

- Implementação da interface **Set**
- Armazena seus elementos em uma tabela hash
  - é uma implementação bastante eficiente
- Eliminando duplicatas em uma coleção:

```
Collection<Type> c;

// Cria collection noDups (sem duplicações) a partir da Collection c

Collection<Type> noDups = new HashSet<Type>(c);    // ou

Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

Exemplo HashSet

```
public class FindDups {
  public static void main(String args[]) {
    Set<String> s = new HashSet<String>();
    for (int i=0; i<args.length; i++)              // Com for-each fica ....
      String a = args[i];                          // for( String a : args )
      if ( ! s.add(a) )
        System.out.println("Duplicata encontrada: "  + args[i]);
      System.out.println(s.size()+"palavras distintas encontradas:"+s);
    }
}
```

- Execução:
  - **% java FindDups i came i saw i left**

  Duplicata encontrada: i
  Duplicata encontrada : i
  4 palavras distintas encontradas: [came, i, left, saw]

Operações sobre conjuntos

- União

  Set<Type> união = new HashSet<Type>(s1);

  união.addAll(s2);

- Interseção

  Set<Type> interseção = new HashSet<Type> (s1);

  interseção.retainAll(s2);

- Diferença

  Set<Type> diferença = new HashSet<Type> (s1);

  diferença.removeAll(s2);

Exemplo Bulk Operation - HashSet

```
import java.util.*;
public class FindDups2 {
  public static void main(String args[]) {
    Set<String> uniques = new HashSet<String> ();
    Set<String> dups = new HashSet<String> ();
    for (int i=0; i<args.length; i++)          // Com for-each fica ....
      String a = args[i];                      // for( String a : args )
      if ( ! uniques.add(a ) dups.add(a);
    uniques.removeAll(dups);                   // Destructive set-difference !
    System.out.println("Unique words: " + uniques);
    System.out.println("Duplicate words: " + dups);
  }
}
```

- **Execução:**
  - % java FindDups2 i came i saw i left

  Unique words: [came, left, saw]

  Duplicate words: [i]

Uma **boa** prática de programação

- O código do exemplo refere-se à coleção pelo tipo de sua interface (**Set<String>**) e não pelo tipo de sua implementação (**HashSet<String>**)
  - caso seja necessário trocar a implementação utilizada, apenas o construtor da coleção precisa ser alterado
  - impede o uso de operações disponíveis apenas em uma implementação específica, evitando erros resultantes da troca dessa implementação

Exemplo Symetric Difference

- A Diferença simétrica entre 2 conjuntos é definida como o conjunto dos elementos contidos em ambos mas não na interseção
- A ∆ B = A ∪ B – A ∩ B

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);          // symdiff = A ∪ B

Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2);                 // tmp = A ∩ B

symmetricDiff.removeAll(tmp);      // symdiff = symdiff - tmp
```


Interface **List**

- Representa uma sequência ordenada de elementos (*ordered collection*)
  - pode conter elementos duplicados
- Permite controlar a posição de inserção de um elemento e asssar elementos por sua posição
- A maior parte dos algoritmos da classe **Collections** são aplicados a listas
- As classes **Vector, ArrayList e LinkedList** implementam a interface **List**

Métodos de **List**

```
public interface List<E> extends Collection<E> {
  // Positional Access
  E get(int index);
  E set(int index, E element);                          // Optional
  boolean add(E element);                               // Optional
  void add(int index, E element);                       // Optional
  E remove(int index);                                  // Optional
  boolean addAll(int index, Collection<? Extends E> c); //Optional
  // Search
  int indexde(Object o);
  int lIndexde(Object o);
  // Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
  // Range-view
  List<E> subList(int from, int to);
}
```


Exemplos List

- Swap 2 elementos de List

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

- Shuffle, permutação randomica dos elementos de List

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

Exemplos List

- Exemplo:

```
public class Shuffle {
   public static void main(String[] args) {
      List<String> list = new ArrayList<String>();
      for (String a : args) list.add(a);
      Collections.shuffle(list, new Random());
      System.out.println(list);
   }
}
```

- Exemplo – mais eficiente !!!

```
public class Shuffle {
   public static void main(String[] args) {
      List<String> list =  Arrays.asList(args);    // asList static factory
      Collections.shuffle(list);                    // não copia o array !!!
      System.out.println(list);
   }
}
```

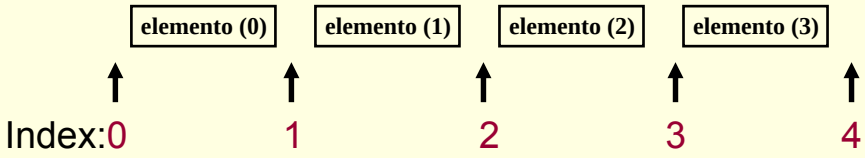

Iteradores de Listas

- O iterador de uma lista obedece à sua sequência
- Listas possuem também iteradores especiais (**ListIterators**) que permitem:
  - percorrer uma lista nos dois sentidos
  - modificar a lista sendo percorrida
  - obter a posição corrente do iterador

Métodos de **ListIterator**

```
public interface ListIterator<E> extends Iterator<E> {
  // Moving forward
  boolean hasNext();
  E next();
  // Moving backward
  boolean hasPrevious();
  E previous();
  // Indexed access
  int nextIndex();
  int previousIndex();
  // Operations on current position
  void remove();          // Optional - remove current position
  void set(E o);          // Optional - overwrites l element returned
  void add(E o);          // Optional – add before current position
}
```


Posicionamento do Iterador



- **nextIndex** retorna o índice do elemento que será retornado na próxima chamada a **next**
- **previousIndex** retorna o índice do elemento que será retornado na próxima chamada a **previous**

Exemplo **ListIterator**

- **Exemplos**

```
// Percorrendo uma lista de trás para frente

for( ListIterator<Type> i=list.listIterator(l.size());  i.hasPrevious(); ) {
   Type e = i.previous();
   ...
}

//Pesquisa da posição de elemento – List.indexde

public int indexde(E e) {
  for (ListIterator i = listIterator(); i.hasNext(); )
    if ( e==null ? i.next()==null : e.equals(i.next()) )
      return i.previousIndex();
  return -1;                                  // Object not found !!!
}
```


Método List.add

- O método add adiciona o novo elemento na lista imediatamente antes da posição corrente do cursor.
- O método abaixo troca todas as ocorrencias de um valor especificado por uma nova lista especificada em newVals

```
public static <E> void replas( List<E> list, E val,
                                List<? extends E> newVals) {
  for (ListIterator<E> it = list.listIterator(); it.hasNext(); ) {
    if (val == null ? it.next() == null : val.equals(it.next())) {
      it.remove();
      for (E e : newVals)
        it.add(e);
    }
  }
}
```

Implementação: **ArrayList**

- Implementação da interface **List**
- Armazena seus elementos em um *array* que cresce dinamicamente
  - é equivalente a **Vector**, sem sincronização
- Implementa todas as operações opcionais
- Ótimo desempenho para assso
  - listas sem muitas modificações no início e meio

Métodos de **ArrayList**

```
public ArrayList(int initialCapacity)
public ArrayList(Collection c)
public ArrayList()
public void trimToSize()
public void ensureCapacity(int minCapacity)
```

Exemplo Baralho

```
class Deal {
 public static void main(String args[]) {
 int numMãos = Integer.parseInt(args[0]); int cartasPorMão = Integer.parseInt(args[1]);
 // Constroi um baralho com as 52 cartas
 String[] naipes = new String[] {"espadas","copas","ouro","paus"};
 String[] seq=new String[]{"ás","2","3","4","5","6","7","8","9","10",
                           "valete","dama", "rei"};
 List<String>  baralho = new ArrayList<String> ();
 for (int i=0; i<naipes.length; i++)
   for (int j=0; j<seq.length; j++)
     baralho.add(seq[j] + " de " + naipes[i]);
  Collections.shuffle(baralho);
 for (int i = 0; i < numMãos; i++)
   System.out.println(escolheMão(baralho, cartasPorMão));
}
```


Exemplo (cont)

```
public static <E>  List<E>  escolheMão(List<E> baralho, int n) {
   int tamanhoBaralho = baralho.size();
   List<E>  visão = baralho.subList(tamanhoBaralho-n,
                                    tamanhoBaralho);
   List<E>  mão = new ArrayList<E>(visão);
   visão.clear();
   return mão;
}
```

- **% java Deal 4 5**

[8 de copas, valete de espadas, 3 de espadas, 4 de espadas, rei de ouros]
[4 de ouros, ás de paus, 6 de paus, valete de copas, rainha de copas]
[7 de espadas, 5 de espadas, 2 de ouros, rainha de ouros, 9 de paus]
[8 de espadas, 6 de ouros, as de espadas, 3 de copas, as de copas]

Interface **Queue**

- Uma coleção para armazenar elementos antes de processa-los. Prove operacoes de inserção, extração e inspeção de objetos.
  - corresponde às abstrações de filas, pilhas e deque
- Queues ordenam os elementos seguindo um criterio de ordenação (em FIFO, LIFO ou PriorityQueues) fornecida através da interface comparator na criação da Queue
- Exemplos:
  - Pilha para calculadora polonesa (estilo HP)
  - Filas de processos prontos em um SO

Métodos de **Queue**

```
public interface Queue<E> extends Collection<E> {
  boolean add( E e );      // throws exception if the operation fails
  boolean offer( E e );    // returns a special value (null or false)

  E remove();              // throws exception
  E poll();                // returns a special value (null or false)

  E element();             // throws exception
  E peek();                // returns a special value (null or false)
}
```

- **Bounded Queues – restringem o numero de elementos da Queue. Ocorrem em java.util.concurrent**

Métodos de Queue

- add – lança IllegalStateException quando violar as restricoes de capacidade da Queue;
- offer – somente usado para bounded queues a diferença em relação ao add é que retorna false em caso de falha;
- remove e poll – removem e retornam o head (nó cabeça) da queue. Quando a queue esta vazia remove lança NoSuchElementException, enquanto poll retona null;
- element e peek – retornam mas não removem o head da queue. Quando a queue esta vazia element lança NoSuchElementException, enquanto peek retona null.
- Implementações de queue geralmente não permitem inserção de elementos nulos. LinkedList é uma exceção !

Implementação: **LinkedList**

- Implementação da interface **Queue**
- Implementa todas as operações opcionais da interface **List**
- Provê operações que permitem inserir, obter e remover elementos no início e fim da lista
  - pode ser usada como pilha , fila e deque

Métodos de **LinkedList**

```
public LinkedList(Collection c)
public LinkedList()
public void addFirst(Object o)
public void addLast(Object o)
public Object getFirst()
public Object getLast()throws NoSuchElementException
public Object removeFirst()throws
   NoSuchElementException
public Object removeLast()throws NoSuchElementException
```


Exemplo Queue

- Relógio de Contagem Regressiva

```
public class Countdown {
  public static void main(String[] args) throws InterruptedException {
    int time = Integer.parseInt(args[0]);
    Queue<Integer> queue = new LinkedList<Integer>();
    for (int i = time; i >= 0; i--)
        queue.add(i);
    while ( !queue.isEmpty() ) {
       System.out.println(queue.remove());
       Thread.sleep(1000);
    }
  }
}
```

Algoritmo de HeapSort

- Uso de uma Priority Queue para ordenar uma collection de elementos

```
static <E> List<E> heapSort(Collection<E> c) {
  Queue<E> queue = new PriorityQueue<E>(c);
   List<E> result = new ArrayList<E>();
   while ( !queue.isEmpty() )
       result.add(queue.remove());
    return result;
}
```


Interface **Map**

- Representa um objeto que mapeia chaves em valores
- Um objeto **Map** não pode conter chaves duplicadas
  - cada chave é mapeada para um único valor
- As classes **Hashtable, HashMap, TreeMap e LinkedHashMap** implementam a interface **Map**

Métodos de **Map**

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


Implementação: **HashMap**

- Implementação da interface **Map**
  - provê todas as operações opcionais de **Map**
  - as entradas são armazenadas em uma tabela hash
  - permite o uso de valores **null**
- Não garante ordenação
- É equivalente a **Hashtable** mas sem sincronização

Exemplo Map

- Tabela de frequencia de palavras

```
public class Freq {
  private static final Integer ONE = new Integer(1);
  public static void main(String args[]) {
    Map<String, Integer> m = new HashMap<String, Integer>();
    // Initialize frequency table from command line
    for (int i=0; i < args.length; i++) {
      Integer freq = (Integer)m.get(args[i]);
      m.put(args[i], (freq==null ? ONE : new Integer(freq.intValue() + 1)));
    }
    System.out.println(m.size()+ " palavras distintas encontradas:");
    System.out.println(m);
  }
}
```


Exemplo Map – foreach e autoboxing

- Tabela de frequencia de palavras

```
public class Freq {
  public static void main(String[] args) {
    Map<String, Integer> m = new HashMap<String, Integer>();
    for (String a : args) {                          // foreach e Autoboxing !!!
      Integer freq = m.get(a);    m.put(a, (freq == null) ? 1 : freq + 1);
    }
    System.out.println(m.size() + " distinct words:"); System.out.println(m);
  }
}
```

- % java Freq if it is to be it is up to me to delegate

8 distinct words:

{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

Interface **Map.Entry**

- O metodo **keySet** retorna um conjunto de chaves, sobre o qual se pode usar um iterador
  - for (Iterator<Type> i=m.keySet().iterator(); i.hasNext();)
    System.out.println(i.next());
  - for (KeyType k: m.keySet())
    System.out.println(k);
- Tipo dos elementos do conjunto (**Set**) retornado **entrySet**
  - for (Iterator< Map.Entry<KeyType, ValType>>
    i=m.entrySet().iterator();
    i.hasNext();) { Map.Entry<KeyType,ValType>
    e=(Map.Entry<KeyType,ValType>)i.next();
    System.out.println(e.getKey() + ": " + e.getValue());
    }
  - for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());

Map Algebra

- Saber se um Map é subMap de outro

```
if ( m1.entrySet().containsAll(m2.entrySet()) ) {
 ...
 }
```

- Saber se 2 Map mapeiam os mesmos objetos

```
if ( m1.keySet().equals(m2.keySet()) ) {
 ...
 }
```

- Obter as chaves comuns a 2 Maps

```
Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());
commonKeys.retainAll(m2.keySet());
```

Exemplo Map

- Valida Map com requiredAttrs e apenas com permittedAttrs

```
static <K, V> boolean validate(Map<K, V> attrMap,
                               Set<K> requiredAttrs, Set<K> permittedAttrs) {
   boolean valid = true;
   Set<K> attrs = attrMap.keySet();
   if( !attrs.containsAll(requiredAttrs) ) {
     Set<K> missing = new HashSet<K>(requiredAttrs);
     missing.removeAll(attrs);
     System.out.println("Missing attributes: " + missing);  valid = false;
    }
    if ( !permittedAttrs.containsAll(attrs) ) {
       Set<K> illegal = new HashSet<K>(attrs);
       illegal.removeAll(permittedAttrs);
       System.out.println("Illegal attributes: " + illegal); valid = false;
     }
     return valid;
}
```


Exemplo MultiMap – Anagram groups

- Map cujos Valores sao List

```
public class Anagrams {
  public static void main(String[] args) {
    int minGroupSize = Integer.parseInt(args[1]);
    // Read words from file and put into a simulated multimap
    Map<String, List<String>> m = new HashMap<String, List<String>>();
    try {
      Scanner s = new Scanner(new File(args[0]));
      while (s.hasNext()) {
         String word = s.next(); String alpha = alphabetize(word);
          List<String> l = m.get(alpha);
          if ( l == null ) m.put(alpha, l=new ArrayList<String>());
          l.add(word);
       }
    } catch (IOException e) {
       System.err.println(e); System.exit(1);
    }
```

Exemplo MultiMap – Anagram groups

```
// Print all permutation groups above size threshold
    for ( List<String> l : m.values() )
      if (l.size() >= minGroupSize)
        System.out.println(l.size() + ": " + l);
  }
  private static String alphabetize(String s) {
     char[] a = s.toCharArray();       Arrays.sort(a);
     return new String(a);
  }
}
```

Exemplo: % java Anagrams dicionario.txt 8

9: [estrin, inerts, insert, inters, niters, nitres, sinter, triens, trines]

8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]

Ordenação de Objetos

- Existem duas formas de ordenar objetos
  - a interface **Comparable** provê uma ordenação *natural* para as classes que a implementam

```
public interface Comparable<T> {
   int compareTo(T o) throws ClassCastException
}
```

  - a interface **Comparator** permite controlar a ordenação de objetos

```
public interface Comparator<T> {
   int compare(T o1, T o2) throws ClassCastException
}
```

Exemplo Ordenação - Comparable

```
public class Name implements Comparable {
 private String fName, lName;
 public Name(String fName, String lName) {
  if(fName==null || lName==null) throw new NullPointerException();
  this.fName = fName; this.lName = lName; }
 public String fName(){ return fName; }
 public String lName(){ return lName; }
 public boolean equals(Object o) { // (equals, hasCode) compatíveis
  if ( !(o instancede Name) ) return false;
  Name n = (Name)o;
  return n.fName.equals(fName) && n.lName.equals(lName);
 }
 public int hashCode(){return 31*fName.hashCode()+lName.hashCode();}
 public String toString() {return fName + " " + lName;}
 public int compareTo(Object o) {
  Name n = (Name)o; int lastCmp = lName.compareTo(n.lName);
  return (lastCmp!=0 ? lastCmp : fName.compareTo(n.fName));
 }
}
```


Exemplo Ordenação – Comparable (cont)

```
import java.util.*;
class NameSort {
  public static void main(String args[]) {
    Name n[] = { new Name("John", "Lennon"),
                 new Name("Karl", "Marx"),
                 new Name("Groucho", "Marx"),
                 new Name("Oscar", "Grouch") };
    List l = Arrays.asList(n);
    Collections.sort(l);
    System.out.println(l);
  }
}
```

- Resultado da execução do sort ! (explique)
  - `[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]`

Exemplo Ordenação – Comparator

- Para sortar objetos que não implementam Comparable ou ordena-los em outra ordem que não a ordenação natural.
- Suponhamos que queiramos ordenar os Empregado pela senioridade. Para isso criaremos uma classe Comparator<Employee>

```
public class Employee implements Comparable<Employee> {
   public Name name() { ... }
   public int number() { ... }
   public Date hireDate() { ... }
   ...
}
```


Exemplo Ordenação – Comparator (cont)

```
public class EmpSort {
  static final Comparator<Employee> SENIORITY_ORDER =  new
   Comparator<Employee>() {
      public int compare(Employee e1, Employee e2) {
         return e2.hireDate().compareTo(e1.hireDate()); }
   };
   // Employee database
   static final Collection<Employee> employees = ... ;
   public static void main(String[] args) {
      List<Employee>e = new ArrayList<Employee>(employees);
      Collections.sort(e, SENIORITY_ORDER); System.out.println(e);
   }
}
```

Interface **SortedSet**

- Representa um conjunto (**Set**) que mantem seus elementos em ordem ascendente
  - ordenação *natural* ou baseada num objeto **Comparator** fornecido ao construtor
- O iterador de um **SortedSet** percorre o conjunto segundo sua ordenação
- O *array* retornado por **toArray** obedece à ordenação dos elementos

Métodos de **SortedSet**

```
public interface SortedSet<E> extends Set<E> {
  // Range-view
  SortedSet<E> subSet(E fromElement, E toElement);
  SortedSet<E> headSet(E toElement);
  SortedSet<E> tailSet(E fromElement);

  // Endpoints
  E first();
  E last();

  // Comparator access
  Comparator<? super E> comparator();

}
```

Implementação: **TreeSet**

- Implementação da interface **SortedSet**, suportada por uma instância de **TreeMap**
- Construtores:

```
public TreeSet()
public TreeSet(Collection c)
public TreeSet(Comparator c)
public TreeSet(SortedSet s)
```


Interface **SortedMap**

- Representa um mapeamento cujas entradas são mantidas ordenadas ascendentemente pelas chaves
  - ordenação *natural* ou baseada num objeto **Comparator** fornecido ao construtor
  - usado como dicionários e listas telefonicas

Métodos de **SortedMap**

```
public interface SortedMap<K, V> extends Map<K, V> {
  // Range-view
  SortedMap<K, V> subMap(K fromKey, K toKey);
  SortedMap<K, V> headMap(K toKey);
  SortedMap<K, V> tailMap(K fromKey);

  // Endpoints
  K firstKey();
  K lastKey();

  // Comparator access
  Comparator<? super K> comparator();

}
```


Exemplo

```
SortedMap m = new TreeMap();
m.put("Sneezy", "common cold");
m.put("Sleepy", "narcolepsy");
m.put("Grumpy", "seasonal affective disorder");
System.out.println(m.keySet());
System.out.println(m.values());
System.out.println(m.entrySet());
```

- Resultado da execução

```
[Grumpy, Sleepy, Sneezy]
[seasonal affective disorder, narcolepsy, common cold]
[Grumpy=seasonal affective disorder, Sleepy=narcolepsy, Sneezy=common cold]
```

Algoritmos

- Métodos estáticos providos pela classe **Collections**
  - primeiro parâmetro é a coleção sobre a qual a operação deve ser realizada
  - o segundo parâmetro quando usado é uma referencia para um Comparator
- A maioria dos algoritmos opera sobre objetos do tipo **List**

Algoritmos com List

1. sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
2. shuffle — randomly permutes the elements in a List.
3. reverse — reverses the order of the elements in a List.
4. rotate — rotates all the elements in a List by a specified distance.
5. swap — swaps the elements at specified positions in a List.
6. replaceAll — replaces all occurrences of one specified value with another.
7. fill — overwrites every element in a List with the specified value.
8. copy — copies the source List into the destination List.
9. binarySearch — searches for an element in an ordered List using the binary search algorithm.
10. indexOfSubList — returns the index of the first sublist of one List that is equal to another.
11. lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.

Alguns algoritmos

```
public static void sort (List l)
public static void sort (List l, Comparator c)

public static int binarySearch (List l, Object key)
public static int binarySearch (List l,Object key, Comparator c)

public static void reverse (List l)
public static void shuffle (List l)

public static Object min (Collection col)
public static Object min (Collection col, Comparator comp)
public static Object max (Collection col)
public static Object max (Collection col, Comparator comp)
```


Exemplo

```
// Make a List de all permutation groups above size threshold
List winners = new ArrayList();
for(Iterator i=m.values().iterator(); i.hasNext(); ) {
  List l = (List)i.next();
  if( l.size() >= minGroupSize) winners.add(l);
}

// Sort permutation groups according to size
Collections.sort(winners, new Comparator() {
  public int compare(Object o1, Object o2) {
   return ((List)o2).size() - ((List)o1).size(); }
});

// Print permutation groups
for (Iterator i=winners.iterator(); i.hasNext(); ) {
  List l = (List)i.next();
  System.out.println(l.size() + ": " + l);
}
```

Implementações *Wrapper*

- São implementações que “encapsulam” uma coleção, adicionando (ou removendo) alguma funcionalidade
- Essas implementações são *anônimas*
  - a classe **Collections** provê métodos estáticos (fábricas) que retornam instâncias dessas implementações

Coleções Sincronizadas

```
public static Collection synchronizedCollection(Collection c);
public static Set synchronizedSet(Set s);
public static List synchronizedList(List list);
public static Map synchronizedMap(Map m);

public static SortedSet synchronizedSortedSet(SortedSet s);
public static SortedMap synchronizedSortedMap(SortedMap m);
```

- Exemplos

```
List l = Collections.synchronizedList (new ArrayList());// = Vector !

Collection c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
  Iterator i = c.iterator(); // Must be in synchronized block!
  while( i.hasNext() )
    foo(i.next());
}
```

Exemplos

```
Map m = Collections.synchronizedMap(new HashMap());
  ...
Set s = m.keySet();  // Needn't be in synchronized block
  ...
synchronized(m) {          // Synchronizing on m, not s!
  Iterator i = s.iterator(); // Must be in synchronized block
  while(i.hasNext())
    foo(i.next());
}
```


Coleções não modificáveis

```
public static Collection unmodifiableCollection(Collection c);

public static Set  unmodifiableSet(Set s);
public static List unmodifiableList(List list);
public static Map  unmodifiableMap(Map m);

public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);
```

- Exemplos

```
List l = Arrays.asList(new Object[size]);
List l = new ArrayList(Collections.nCopies(1000, null));
```