estgoh
Politécnico de Coimbra

Programação
Programação III

Estruturas de Dados Dinâmicas

Marco Veloso
marco.veloso@estgoh.ipc.pt

Agenda

Estruturas de Dados Dinâmicas

Introdução à Linguagem Java
- Paradigmas de Programação
- Linguagem Java

Programação Orientada a Objectos
- Objectos
- Classes
- Heranças
- Polimorfismo

Tratamento de Excepções

Estruturas de dados
- Tabelas unidimensionais
- Tabelas multidimensionais
- Vectores
- Dicionários (Hashtables)
- Collections

Ficheiros
- Manipulação do sistema de ficheiros
- Ficheiros de Texto
- Ficheiros Binários
- Ficheiros de Objectos
- Leitura de dados do dispositivo de entrada

Vectores & ArrayList

A **limitação principal das tabelas** é a necessidade de **indicar o seu tamanho máximo no momento da sua criação**

O Java inclui a classe **`Vector`** que tem semelhanças com as tabelas, mas permite ultrapassar aquela limitação

Existem três diferenças fundamentais entre tabelas e vectores:

– **Os vectores podem crescer ou decrescer de tamanho** em função das necessidades do programa

– **Os vectores apenas podem armazenar objectos**, pelo que não é possível ter um vector contendo tipos simples (**`int`**, **`double`**, etc.)

– Contrariamente às tabelas (elementos todos do mesmo tipo), os **vectores podem conter em simultâneo objectos de classe diferentes** (desde que sejam objectos)

Criação de vectores

Estruturas de Dados Dinâmicas

Os vectores são objectos da classe **`Vector`**, incluída na biblioteca **`java.util`**, que tem que ser **importada** para qualquer programa que use vectores (**`import java.util.Vector;`**)

Um vector pode ser criado como qualquer **objecto**, ou seja, com uma **declaração** e uma **inicialização** com o elemento **`new`**:

```
Vector lista = new Vector();
```

Daqui resulta um objecto da classe **`Vector`** (genérico) referenciado por **`lista`**, que pode conter um conjunto de objectos variável (aumenta automaticamente à medida que são adicionados novos elementos) e que tem definidos vários comportamentos que permitem manipular esse conjunto

Se não for especificado o tamanho do vector, o construtor vazio constrói uma estrutura com espaço para dez elementos, sem conteúdo.

(http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html)

Comportamentos da classe *Vector*

Alguns comportamentos da classe **Vector**:

Comportamento	Função
`boolean` **`add`** `(objecto)`	Adiciona o objecto no fim do vector
`boolean` **`add`** `(indice, objecto)`	Insere o objecto na posição indicada por índice
`void` **`addElement (objecto)`**	Adiciona um objecto no fim do vector, incrementando (duplicando) a capacidade da estrutura, se necessário
`Object` **`elementAt (indice)`**	Devolve o objecto colocado na posição indicada por índice
`void` **`clear`** `()`	Elimina todos os elementos do vector
`int` **`indexOf (objecto)`**	Devolve o índice da primeira ocorrência de objecto no vector (-1 caso não encontre)
`Vector` **`remove`** `(indice)`	Remove o objecto colocado na posição dada por índice
`boolean` **`remove`** `(objecto)`	Remove o objecto do vector
`void` **`setElementAt`** `(objecto, indice)`	Substitui o objecto da posição índice pelo objecto dado
`int` **`size ()`**	Devolve o **número de elementos** do vector (método **`capacity()`** devolve a **capacidade da estrutura**)

Aplicação de vectores

A classe **`Vector`** poderia ser utilizada para criar uma classe **`Turma`** que vai incluir um conjunto de estudantes (representados pela classe **`Estudante`**).

A nova classe possui um atributo da classe **`Vector`**:

```
private Vector lista;          //declaração
```

O construtor da classe cria o vector, guardando o seu endereço em lista:

```
public Turma() {
        lista = new Vector(); //inicialização
}
```

Caso nada seja indicado o vector será criado com espaço para 10 elementos por defeito. Na inicialização do vector (chamada do construtor) é possível definir a dimensão inicial:

```
        lista = new Vector(5);
```

Será útil incluir um **método que adicione** um novo estudante à turma. A chamada do construtor da classe **`Estudante`** faz com que sejam pedidos ao utilizador os dados do novo estudante:

```
public void juntaEstudante() {
        Estudante e = new Estudante();
        lista.addElement (e);
}
```

As duas instruções deste método podem ser substituídas por:

```
lista.addElement ( new Estudante() );
```

Para o nosso problema, método `imprimeTurma()` terá que **iterar sobre todos os elementos** do vector:

```
public void imprimeTurma() {
    Estudante temp;

    for (int i=0; i < lista.size(); i++) {
        temp = (Estudante) lista.elementAt(i);
        temp.mostraEstudante();
    }
}
```

As duas linhas do ciclo podem ser substituídas por uma única instrução, evitando a necessidade de criar um objecto `Estudante` (objecto `temp`):

```
    for (int i=0; i < lista.size(); i++) {
        ((Estudante) lista.elementAt(i)).mostraEstudante();
    }
```

Para percorrer todos os elementos de um **Vector** pode-se recorrer a um ciclo **`for`**, semelhante ao procedimento usado em tabelas:

```
Vector lista = new Vector();
//…
lista.addElement(new String("Hello World"));
//…
for (int i=0; i < lista.size(); i++) {
      String temp = (String) lista.elementAt(i);
      System.out.println(temp);
}
```

Notar:

- a utilização do **método `size()` para obter a dimensão real do vector**

- o ***cast* para indicar a classe dos elementos lidos** do vector (um vector pode ter objectos de classes diferentes)

Em alternativa, poderemos igualmente usar a forma contraída **`foreach`** para percorrer o **Vector**

```
Vector lista = new Vector();
//…
lista.addElement(new String("Hello World"));
//…
for (String temp : lista) {
      System.out.println(temp);
}
```

No entanto, o uso de ciclos **`for`** e **`foreach`** não são as abordagens mais eficientes para percorrer estruturas de dados dinâmicas.

Iteração de Vectores (*Enumeration*)

Estruturas de Dados Dinâmicas

Para iterar sobre um vector podemos recorrer, como alternativa mais eficiente a um ciclo **`for`**, a duas classes: **`Enumeration`** e **`Iterator`**.

A classe **`Enumeration`** disponibiliza dois métodos: **`hasMoreElements()`**, que devolve *true* se a estrutura **contém mais elementos**, e **`nextElement()`**, que **devolve o próximo elemento** da estrutura.

Primeiro é necessário obter uma *enumeration* (**`elements()`**), uma estrutura que vai disponibilizar os elementos contidos no vector, e a seguir percorrer essa estrutura:

```
import java.util.Enumeration;
...
Vector lista = new Vector();
...
Enumeration en = lista.elements();                    // obtém estrutura
while (en.hasMoreElements()) {                        // verifica próximo elemento
        String elemento = (String) en.nextElement();// obtém o próximo elemento
}
```

Sendo necessário proceder à **importação** das respectiva classe (**`import java.util.Enumeration`**)

O **`Enumeration`** é uma **estrutura rápida para pesquisas sequências**, usando menos memória, mas menos seguro a garantir a integridade dos dados.

Note-se que sempre que se obtém um elemento do vector, este é do tipo **`Object`**, sendo **necessário fazer um *cast* para o tipo de dados pretendido**

(http://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html)

Iteração de Vectores (*Enumeration*)

Estruturas de Dados Dinâmicas

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerarVector {

	public static void main(String[] args) {
		Vector v = new Vector();
		v.add("elemento1");
		v.add("elemento2");
		v.add("elemento3");

		Enumeration listaElementos = v.elements();

		System.out.println("elementos do vector:");

		while(listaElementos.hasMoreElements())
			System.out.println((String) listaElementos.nextElement());
	}
}
```

Iteração de Vectores (*Iterator*)

A classe **`Iterator`** (mais recente, associada às *collections*) apresenta um funcionamento análogo

De igual forma, a classe disponibiliza dois métodos: **`hasNext()`**, que devolve *true* se a estrutura **contém mais elementos**, e **`next()`**, que **devolve o próximo elemento da estrutura**

Sendo primeiro necessário obter um *iterator* (**`iterator()`**), uma estrutura que vai disponibilizar os elementos contidos no vector, e a seguir percorrer essa estrutura:

```
import java.util.Iterator;
...
Vector listaElementos = new Vector();
...
Iterator it = listaElementos.iterator();          // obtém estrutura
while (it.hasNext()) {                            // verifica próximo elemento
        String elemento = (String) it.next();     // obtém o próximo elemento
}
```

É sempre necessário proceder à **importação** da respectiva classe (**`import java.util.Iterator;`**)

O **`Iterator`** é uma **estrutura mais lenta em pesquisas sequências**, mas **mais segura**, **salvaguardando a concorrência entre *threads***

Note-se que sempre que se obtém um elemento do vector, este é do tipo **`Object`**, sendo **necessário fazer um *cast* para o tipo de dados pretendido**

(http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html)

Iteração de Vectores (*Iterator*)

Estruturas de Dados Dinâmicas

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

public class EnumerarVector {

	public static void main(String[] args) {
		Vector v = new Vector();
		v.add("elemento1");
		v.add("elemento2");
		v.add("elemento3");

		Iterator listaElementos = v.iterator();

		System.out.println("elementos do vector:");

		while(listaElementos.hasNext())
			System.out.println((String) listaElementos.next());
	}
}
```

Iteração de Vectores

Estruturas de Dados Dinâmicas

A necessidade de realizar uma **conversão explicita** (*cast*) no acesso a um elemento de um vector advém do facto de um **vector poder suportar diferentes tipos de objectos simultaneamente**

Na generalidade das situações em que se recorre a um vector, os objectos armazenados possuem o mesmo tipo

Neste cenário podemos **indicar o tipo de objectos que um vector irá suportar**, eliminado a necessidade de conversões:

```
Vector <String> listaStrings = new Vector <String> ();
```

Durante a iteração, não existe necessidade de conversões, pelo que também deveremos informar a classe de iteração do tipo de dados a obter:

```
Enumeration <String> enumString = listaStrings.elements();

while(enumString .hasMoreElements()) {
        String elemento = enumString.nextElement();
}
```

Como o Vector é definido como sendo uma estrutura de Strings, não é necessário converter o object

Continua a ser possível **percorrer os elementos** da estrutura recorrendo a um **ciclo simples**:

```
Vector<String> tab = new Vector<String>(2);

tab.add("a");

tab.add("b");
for (int index=0; index < tab.size(); index++) {
      System.out.println(tab[index]);
}
```

Alternativamente, o ciclo pode ser definido da seguinte forma:

```
for(String var: tab) {
      System.out.println(var);
}
```

Dicionários (*Hashtables*)

Uma **limitação dos vectores** surge na **pesquisa de elementos**, pois tal só é possível fornecendo uma **cópia exacta do objecto** a pesquisar

Os **dicionários** (*Hashtables*) **facilitam a pesquisa, inserção, a alteração e remoção de elementos numa lista**, por ser tudo efectuado a partir da chave única associada a cada elemento

Os dicionários **não utilizam índices numéricos** para aceder aos seus elementos, mas sim **chaves únicas associadas a cada elemento** armazenado

A **pesquisa é feita sobre a chave** e não sobre os objectos completos

Uma vez encontrada a chave procurada, utiliza-se um método para obter o objecto que lhe está associado

As **chaves são objectos de qualquer tipo**, embora normalmente sejam objectos da classe *String*

Comportamentos da classe *Hashtable*

Os **dicionários** são implementados pela classe **`Hashtable`** incluída na biblioteca **`java.util`** (**`import java.util.Hashtable;`**) ideais para **pesquisas frequentes**

Para criar um objecto desta classe:

```
Hashtable dicionario = new Hashtable();
```

Se não for especificado o tamanho do dicionário, o construtor vazio constrói uma estrutura com espaço para 11 elementos, sem conteúdo, aumentando automaticamente o seu tamanho quando atingir ¾ da sua capacidade

Cada elemento que se coloca na *hashtable* é composto por 2 objectos:

- **chave** (que permite aceder ao outro objecto) – ***key***
- **objecto** que se pretende guardar – ***value***

(http://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html)

Comportamentos da classe *Hashtable*

Alguns métodos úteis:

Função	Comportamento
`put (chave, objecto)`	Adiciona ao dicionário a associação chave-objecto
`get (chave)`	Devolve o objecto associado com chave ou *null* caso esta não exista
`remove (chave)`	Remove do dicionário o elemento associado a *chave*
`containsKey (chave)`	Indica se o objecto fornecido é já usado como chave
`clear ()`	Elimina todos os elementos (objectos) do dicionário
`isEmpty()`	indica se a *hashtable* está vazia (não tem chaves)
`size ()`	Devolve o número de elementos do dicionário

Os **dicionários são particularmente adequados** quando há que fazer **pesquisas frequentes** nos dados armazenados

Por exemplo, para pesquisar o objecto representativo de um dado estudante:

```
public void procuraEstudante(String nome) {
        String chave = nome.toUpperCase();
        Estudante estud = (Estudante) dicionario.get(chave);
        if (estud != null) {
                estud.mostraEstudante();
        else
                System.out.println("Aluno inexistente...");
        }
}
```

Iteração sobre os elementos

Por outro lado, a **iteração sobre todos os elementos de um dicionário é mais difícil**, pois **não há índices**

A classe **`Enumeration`** disponibiliza igualmente dois métodos (como no caso dos vectores): **`hasMoreElements()`**, que devolve *true* se a estrutura **contém mais elementos** (chaves), e **`nextElement()`**, que **devolve o próximo elemento** da estrutura,

Neste caso **a estrutura é uma lista de chaves** (e não dos objectos). Com cada chave podemos obter o respectivo elemento através do método **`get()`** aplicado ao dicionário

Assim, primeiro é necessário obter uma *enumeration* (método **`key()`**, enquanto em vectores é usado o método **`elements()`**), uma estrutura que vai disponibilizar as **chaves do dicionário**. Percorrendo essa estrutura temos acesso a cada chave, que por sua vez permite **obter do dicionário o respectivo elemento**:

```
Hashtable dicionario = new Hashtable();
Enumeration en = dicionario.keys();
while (en.hasMoreElements()) {
   String    chave = (String) en.nextElement();
   Estudante valor = (Estudante) dicionario.get(chave);
   System.out.println(valor);
}
```

Iteração sobre os elementos

Um método (aplicável a vectores e dicionários) para imprimir os dados de todos os elementos de uma lista (e.g. Estudantes) teria que usar uma classe auxiliar (tal como apresentado para os vectores), a *Enumeration*:

```
public void imprimeTurma() {
  if (dicionário != null && dicionario.size() > 0) {
    Enumeration en = dicionario.keys();
    while (en.hasMoreElements()) {
       String chave = (String) en.nextElement();
       Estudante estudante = (Estudante) dicionario.get(chave);
       estudante.mostraEstudante()
    }
  } else
    System.out.println ("Lista de alunos vazia...");
}
```

Iteração sobre os elementos

Estruturas de Dados Dinâmicas

```
import java.util.Hashtable;
import java.util.Enumeration;

public classSiglasMatriculas {
     public static void main(String[] args) {
            //…
            //enumerar todos os elementos do dicionário
            //criar uma "lista" com as chaves do dicionário
            Enumeration todasChaves = tab.keys();

            while(todasChaves.hasMoreElements()) {
                  chave = (String) todasChaves.nextElement();
                  localidade = (String) tab.get(chave);
                  System.out.println(chave+ " = "+ localidade);
            }
     }
}
```

Collections

Uma ***collection*** (também designada de *container*, ou colecção/contentor de elementos) é simplesmente um **objecto que agrupa múltiplos elementos numa única unidade**, de representação e processamento devidamente organizada, possuindo propriedades de estrutura e funcionamento próprias.

As *collections* são usadas para **guardar**, **obter** e **manipular dados**, bem como transmitir dados de um método para outro, representando dados que formam grupos naturais

As versões iniciais do Java já continham *collections*, sob a forma de **Vectores**, **Dicionários** (*Hashtables*) e **Tabelas** (*Arrays)*

No entanto, estas versões iniciais continham implementações de *collections*, mas não uma *collection framework*

Uma ***collection framework*** é uma arquitectura para representação e manipulação de *collections*. Todas as *collections framework* contêm 3 características:

- **Interface**: tipos de dados abstractos representando *collections*;

- **Implementação**: implementação concreta das interfaces;

- **Algoritmo**: método que produz acções computacionais como **pesquisas** e **ordenações** nos objectos que implementam as *collections*. Estes métodos são **polimórficos**;

Existem **4 estruturas** que servem de base como *interface* das ***Collections***, seguindo a sintaxe:

Collection<E> (colecção de elementos do tipo *E*, *E* é um tipo ou uma classe)

http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Estruturas:

Set	***Set<E>***	http://docs.oracle.com/javase/8/docs/api/java/util/Set.html
List	***List<E>***	http://docs.oracle.com/javase/8/docs/api/java/util/List.html
Map	***Map<K,V>***	http://docs.oracle.com/javase/8/docs/api/java/util/Map.html
Queue	***Queue<E>***	http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

Uma interface define a estrutura de uma classe, mas não pode ser instanciado (não permite o uso da *keyword* `new`()). É assim necessário criar classes que representem (**implementam**, necessitam da *keyword* `implements` na definição) estas interfaces e que possam ser instanciadas.

Por exemplo, a classe ***Vector*** é uma implementação da **interface *List***, enquanto a classe ***Hashtable*** é uma implementação da **interface *Map***

A interface ***Collection*** é a raiz de toda a hierarquia. Representa um **grupo de objectos**, definidos como ***elements***

Set – (**conjunto**) é um conjuntos de objectos **sem ordem** que **não podem conter elementos duplicados**

List – (**lista**) é uma sequência ordenada **ordenada** de objectos, **podendo conter elementos duplicados**. Permite **controlo na posição onde um elemento é colocado**, podendo ser **acedido a partir da sua posição** (*index*), por exemplo, como os vectores (***Vector***)

Map – (**mapa**) é um objecto que **mapeia chaves** (*keys*) para valores unívocos (estrutura *chave-valor*), e como tal **não podendo conter chaves duplicadas**, mas podendo possuir diferentes chaves para o mesmo valor por exemplo, como os dicionários (***Hashtable***)

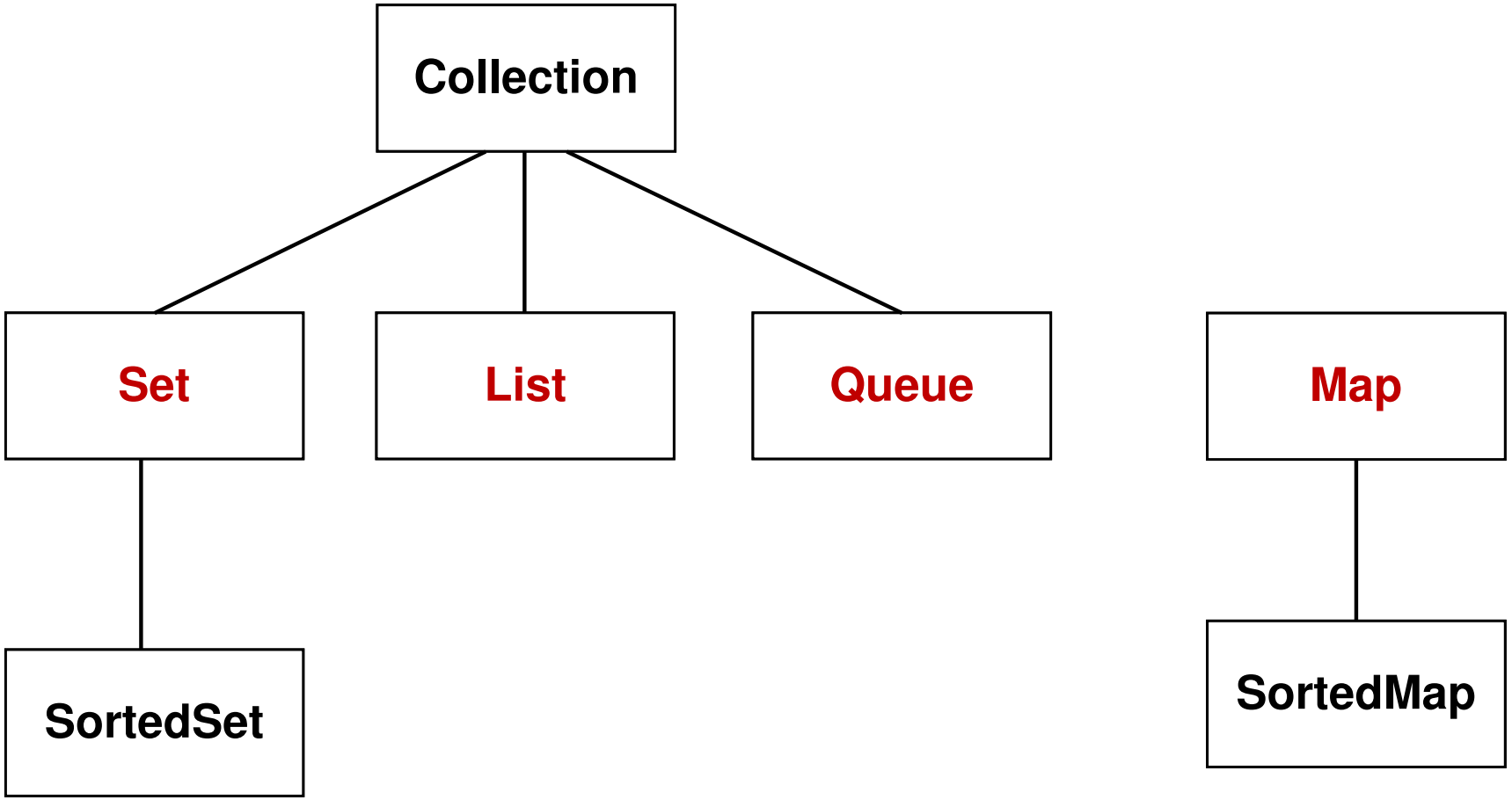
Queue – (**fila**) representa uma **fila**, estrutura linear tipo **First-In-First-Out** (FIFO), contendo métodos próprios de inserção e remoção

SortedSet – é um ***Set*** que **mantêm a ordem os elementos ordenados ascendentemente**, usado, por exemplo, para ordenação alfabética ou numérica

SortedMap – é um ***Map*** que **mantém os elementos ordenados ascendentemente de acordo com as respectivas chaves** (*keys*), usado, por exemplo, para dicionários e listas telefónicas

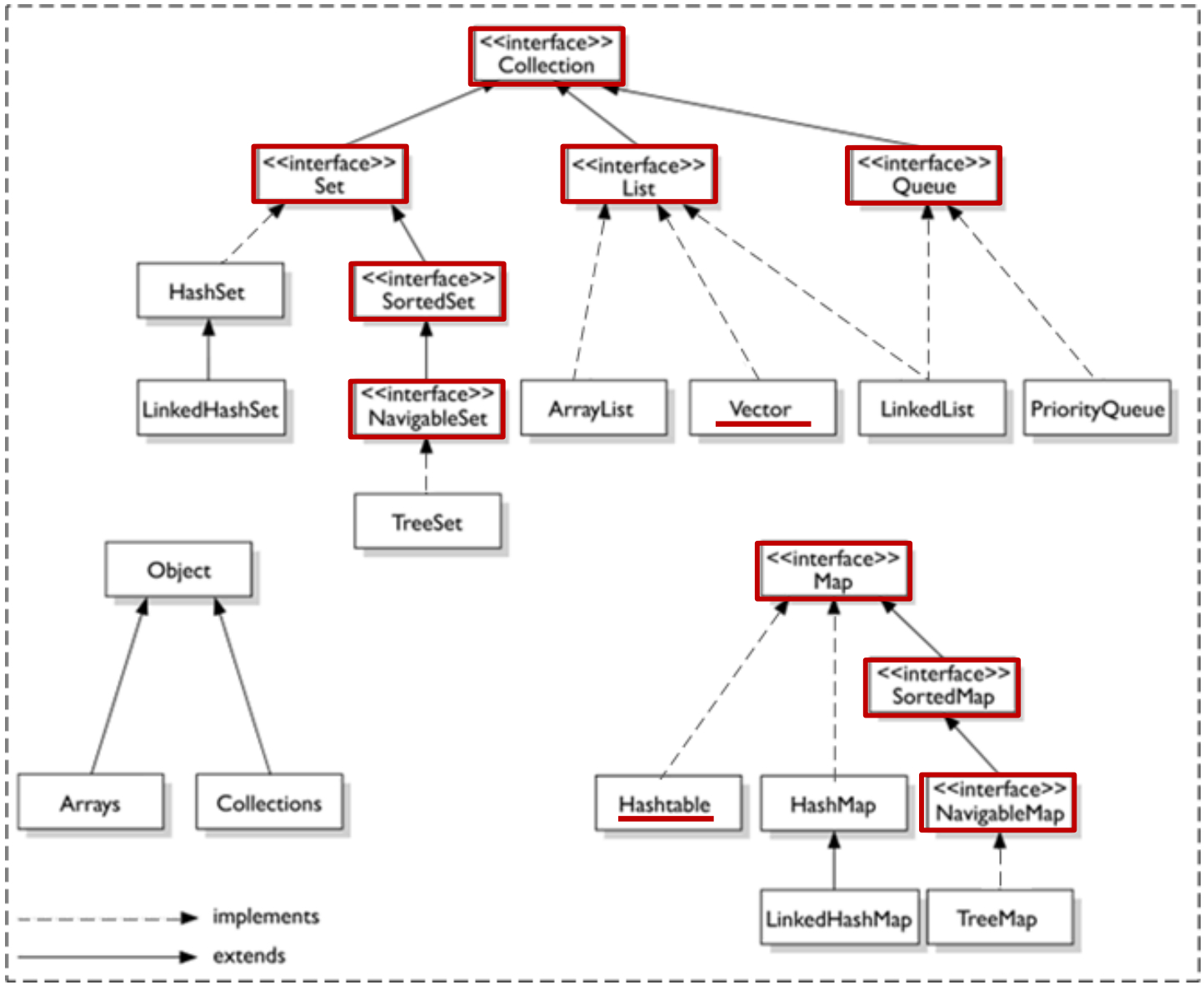
Interfaces

Estruturas de Dados Dinâmicas



Interfaces

Estruturas de Dados Dinâmicas



		Implementações			
		HashTable	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		Linked List
	Map	HashMap		TreeMap	

Propriedades das *Collections*

Como analisamos durante o estudo dos vectores, as *collections* apresentam 3 propriedades:

1. **dimensão pode variar** durante o programa (em função das necessidades);

   (tabelas têm dimensão fixa)

2. só pode haver **colecções de objectos**, não de tipos primitivos (int, double, char,…);

   NOTA: é possível transformar os tipos em objectos.

   (pode haver tabelas de objectos ou tabelas de inteiros, reais, …)

3. os elementos na colecção podem ser de **diferentes tipos** (desde que sejam objectos)

   (todos os elementos de uma tabela são do mesmo tipo)

Collections Interface - List

Estruturas de Dados Dinâmicas

List<E> é uma interface a classe abstracta **AbstractList<E>**

Tipos de listas (implementações):

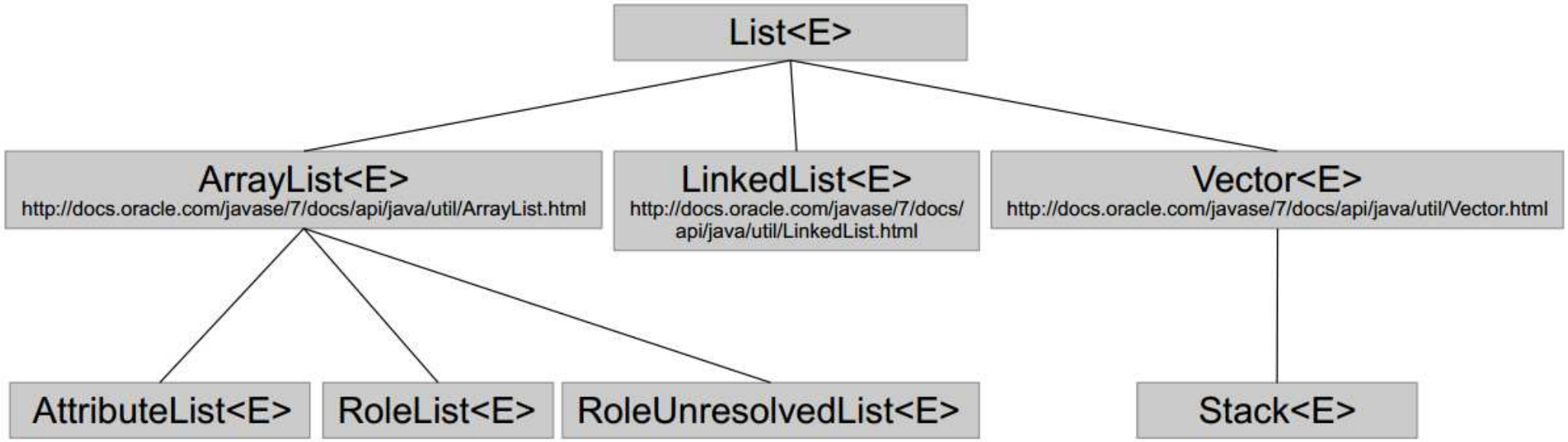
- **ArrayList<E>** - tabela dinâmica não sicronizada
- **Vector<E>** - tabela dinâmica sincronizada
- **Stack<E>** - pilha LIFO (Last-In-First-Out), subtipo de **Vector**
- **LinkedList<E>** - lista duplamente ligada

Podem ter métodos específicos, restantes métodos mantém-se:

```
- boolean   add(objecto)
- boolean   addAll(colecao)
- iterator  iterator()
- boolean   remove(objecto)
- int       size()
- objecto[] toArray()
- boolean   contains(objecto)
```

Collections Interface - List

Estruturas de Dados Dinâmicas



Collections Interface - List

Estruturas de Dados Dinâmicas

Alguns comportamentos da interface **List**, comuns a todas as implementações (**ArrayList**, **LinkedList**, **Vector**) :

Comportamento	Função
`boolean add (objecto)`	adiciona o *objecto* à lista
`boolean add (indice, objecto)`	insere o *objecto* na posição *indice* da lista
`get (indice)`	devolve o objecto que está na posição *indice*
`clear ()`	elimina todos os elementos do vector
`int indexOf (objecto)`	devolve o índice da primeira ocorrência de objecto na lista (-1 caso não encontre)
`Object remove (indice)`	remove o objecto colocado na posição dada por *índice*
`Object remove (objecto)`	retira o *objecto* do vector
`in size ()`	Devolve o número de elementos da lista

Cada implementação (**ArrayList**, **LinkedList**, **Vector**) da interface **List** herda estes métodos e pode definir métodos adicionais

Cada implementação **herda da classe superior os métodos e atributos *public* e *protected***. Por exemplo, a classe **Stack** herda os métodos implementados na classe **Vector** (e.g. **`add()`**), acrescentando as suas próprias implementações (e.g. **`pop()`**)

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector

`add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, iterator, lastElement, lastIndexOf, lastIndexOf, listIterator, listIterator, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toArray, toString, trimToSize`

Methods inherited from class java.lang.Object

`finalize, getClass, notify, notifyAll, wait, wait, wait`

Exemplo de uso de **Vectores**:

```
import java.util.Vector;

public class VectorEstudantes{

    public static void main(String[] args) {
      Vector <Estudante> estudantes = new Vector<Estudante>();
      //Vector estudantes = new Vector();

      estudantes.add(new Estudante("Joao","joao@mail.pt",987654321));
      estudantes.add(new Estudante("Ana","ana@email.com",123456789));

      estudantes.get(0).mostraDados(); //poderia usar elementAt()
      //System.out.println(estudantes.elementAt(0));
      System.out.println("nº de elementos do vector: "+ estudantes.size());
    }
}
```

Exemplo de uso de **ArrayList**:

```
import java.util.ArrayList;

public class ArrayListEstudantes {

    public static void main(String[] args) {
      ArrayList <Estudantes> estudantes = new ArrayList<Estudantes>();


      estudantes.add(new Estudante("Joao","joao@mail.pt",987654321));
      estudantes.add(new Estudante("Ana","ana@email.com",123456789));

      estudantes.get(0).mostraDados();

      System.out.println("nº de elementos do array:  "+ estudantes.size());
    }
}
```

Set<E> é uma interface a classe abstracta **AbstractSet<E>**

Tipos de conjuntos (implementações):

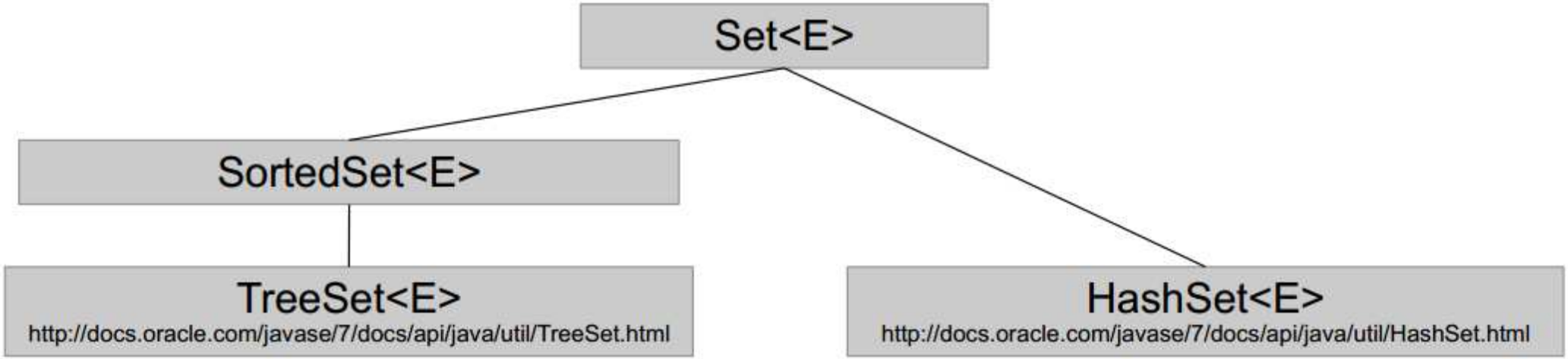
- **HashSet<E>** - tabela de *hashing*
- **TreeSet<E>** - árvore balanceada (para elementos comparáveis)
- **EnumSet<E>** - usa tipos enumerados

Não existe ordem, pelo que **métodos de acesso por índice não existem**, mas restantes métodos mantém-se:

```
- boolean   add(objecto)
- boolean   addAll(colecao)
- iterator  iterator()
- boolean   remove(objecto)
- int       size()
- objecto[] toArray()
- boolean   contains(objecto)
```

Collections Interface – Set

Estruturas de Dados Dinâmicas



Alguns comportamentos da interface **Set**, comuns a todas as implementações (**HashSet**, **TreeSet**, **EnumSet**) :

Comportamento	Função
`boolean` **`add`** `(objecto)`	adiciona o *objecto* à lista
`Object` **`remove`** `(indice)`	remove o objecto colocado na posição dada por *índice*
`int` **`size`** `()`	Devolve o número de elementos da lista
`clear` `()`	elimina todos os elementos do vector
`boolean` **`addAll`** `(Colecção)`	acrescenta todos os elementos da colecção ao conjunto se esses elementos ainda lá não estão (*reunião*)
`boolean` **`containsAll`**`(Colecção)`	devolve verdadeiro se o conjunto contém todos os elementos da colecção (*contém*)
`boolean` **`retainAll`**`(Colecção)`	Mantém no conjunto apenas os elementos que estão na colecção (*intersecção*)

Como na interface anterior, cada implementação da interface **Set** herda estes métodos e pode definir métodos adicionais

```
//Exemplo de implementação de uma HashSet
HashSet <String> hset = new HashSet<String>();
hset.add("C");
hset.add("A");
hset.add("B");

System.out.println("elementos da hashset: "+hset);
//resultado da execucao:
//elementos da hashset: [A, B, C]

//Exemplo de implementação de uma TreeSet
TreeSet <String> conj = new TreeSet<String>();
conj.add("c");
conj.add("a");
conj.add("b");
System.out.println("elementos da treeset: "+conj);
//resultado da execucao:
//elementos da treeset: [a, b, c]
```

Map<K,V>: ***Mapping*** é a **correspondência entre um conjunto de chaves** (***keys*– *K***) e um **conjunto de valores** (***values*– *V***).

Correspondência entre **uma chave** e **um valor** (**1 – 1**)

As chaves:

- são **únicas**
- **formam um conjunto** (**sem ordem** predefinida)
- **duas chaves distintas podem ter o mesmo valor** (mas não é partilhado, é igual) – duas correspondências

remover de um *map* significa **remover o par chave-valor**, ainda que seja apenas indicada a chave

Tipos de *maps* (implementações):

- **EnumMap<K,V>**
- **HashMap<K,V>** - associado a uma **tabela**
- **HashTable<K,V>** - associada a uma **tabela** mas para **concorrência**
- **LinkedHashMap<K,V>** - associado a uma **tabela e lista ligada**
- **TreeMap<K,V>** - **árvore balanceada** para guardaras chaves (mapa ordenado) – ver *TreeSet*

Alguns métodos:

- `void clear()`
- `boolean containsKey(ObjectoC chave)`
- `boolean containsValue(ObjectoV valor)`
- `ObjectoV get(ObjectoC chave)`
- `ObjectoV put(ObjectoC chave,ObjectoV valor)`
- `ObjectoV remove(ObjectoC chave)`
- `int size()`
- `Collection values()` – *devolve um Collection dos objectos*

Iteração em *Collections* (*Enumeration* e *Iterator*)

Para iterar sobre um vector podemos recorrer, como alternativa a um ciclo **`for`**, a duas classes: **`Enumeration`** e **`Iterator`**.

No caso da classe **`Enumeration`** é necessário obter uma *enumeration* que vai disponibilizar os elementos contidos no vector:

```
Vector listaElementos = new Vector();
...
Enumeration enum = listaElementos.elements();
while (enum.hasMoreElements()) {
        String elemento = (String) enum.nextElement();
}
```

A classe **`Iterator`** (mais recente, associada às *collections*) apresenta um funcionamento análogo:

```
Iterator iterator = listaElementos.iterator();
while (iterator.hasNext()) {
        String elemento = (String) iterator.next();
}
```

Sendo necessário proceder à importação das respectivas classes (**`import java.util.Enumeration;`** e **`import java.util.Iterator;`**)

Note-se que sempre que se obtém um elemento do vector, este é do tipo **`Object`**, sendo **necessário fazer um *cast* para o tipo de dados pretendido**

Alternativamente, como indicado anteriormente, para percorrer todos os elementos de uma ***Collection*** (como observado em **Vectores**) pode-se recorrer a um ciclo **`for`**:

```
Vector list = new Vector();
lista.addElement(new String("Hello World"));
//…
for (int i=0; i<lista.size(); i++) {
      String temp = (String) lista.elementAt(i);
      System.out.println(temp);
}
```

Ou na sua forma contraída, através de um **`foreach`**:

```
for (String tmp: list) {
      System.out.println(temp);
}
```

Ordenação

Ordenamento dos objectos

A classe **`Collections`** disponibiliza uma interface com várias implementações (métodos *static*) que podem ser usados em estruturas de dados ou colecções (*arrays*, *arraylists*, listas, vectores, etc.), por exemplo:

- **`binarySearch(…)`** **pesquisa** de elementos usando um algoritmo binário
- **`copy(…)`** **copia** os elementos de uma lista para outra
- **`disjoint(…)`** verifica se duas listas possuem **elementos comuns**
- **`max(…)`** retorna o **maior elemento** da lista (ordem natural)
- **`min(…)`** retorna o **menor elemento** da lista (ordem natural)
- **`reverse(…)`** **inverte** a ordem dos elementos da lista
- **`sort(…)`** **ordena** os elementos de acordo com a ordem natural
- **`swap(…)`** **troca a posição** entre dois elementos

De realçar o método **`sort()`** que permite a **ordenação natural** dos elementos de uma lista

Caso estejamos a lidar com estruturas contendo **elementos simples** (e.g. uma lista de inteiros), que possuem uma ***ordenação natural*** a ordenação poderá ser realizada recorrendo ao método `sort()` da classe `Arrays`.

Para tal é necessário:

(i) **possuir uma *collection*** de dados (*e.g.* um *vector*);

(ii) **invocar o método** `toArray()` para obter uma tabela (*array*) com os elementos da estrutura dinâmica, sendo uma estrutura auxiliar;

(iii) **invocar o método** `sort()` da classe `Arrays` (`Arrays.sort()`), que recebe a tabela como argumento, procedendo à sua ordenação natural.

O ordenamento anterior, sendo simples e eficaz, é **adequado para listagens igualmente simples** (*e.g.* lista de inteiros ou *strings*), existindo um ***ordenamento natural***;

Se estivermos perante uma lista de objectos com várias variáveis (*e.g.* classe **`Estudante`**, caracterizada por **`nome`** e **`número de aluno`**) **não existe o ordenamento natural** (os estudantes tanto podem ser ordenados por nome como por número);

Nesta situação necessitamos de outro procedimento que permita **definir explicitamente a forma como o ordenamento** será realizado;

Recorremos então às classes **`Comparable`** e **`Collections`**

Existem essencialmente duas formas de ordenar objectos:

- **interface *Comparable***: disponibiliza uma **ordenação natural automática** nas classes que a implementam.

  Por exemplo, a classe `Array` disponibiliza o método `void sort(Object [] obj)` que permite ordenar um *array* de objectos.

  Para listas **sem ordenação natural** é possível implementar o método `int compareTo(Object obj)`. A classe instanciada por estes objectos deve implementar esta interface (`implements Comparable`)

- **interface *Comparator***: permite ao programador **completo controlo na ordenação dos objectos**.

  Para o efeito, a classe criada deve implementar esta interface (`implements Comparator`) e deve ser redefinido o método `boolean equals(Object obj)` por forma a ser realizada uma comparação de acordo com o pretendido

(i) A classe que representa o objecto **deve implementar a classe `Comparable`**

```
public class AMinhaClasse implements Comparable {...}
```

(ii) Na classe, ao definirmos a implementação da classe **`Comparable`**, teremos que internamente **implementar o método `int compareTo(Object)`**

Este será o método usado automaticamente para o compilador saber como comparar dois objectos, ou seja, saber qual **o maior, menor, ou se são iguais**, **retornando 1, -1, ou 0**. **O programador é que define essa ordem**:

```
public int compareTo(Object arg0) {
   AMinhaClasse objTmp = (AMinhaClasse ) arg0; // cast explícito
       if (variavelDoObjecto > objTmp.variavelDoObjecto )
           return 1;  // o objecto recebido como argumento é
                                   inferior ao objecto actual
       if (variavelDoObjecto < objTmp.variavelDoObjecto)
           return -1; // é superior

       return 0;      // os objectos são iguais
}
```

Note-se que **o programador é que define o que significa maior, menor ou igual**, **indicando qual a variável que deve ser comparada** e o significado dessa comparação;

(iii) Criar uma estrutura do tipo *collection* (*e.g.* vector) **para armazenar os objectos**:

```
Vector <AMinhaClasse> listaObj;
```

(iv) Para ordenar basta invocar o método `sort()` da classe `Collections` (não da classe `Arrays`) que recebe como argumento uma estrutura dinâmica;

A ordenação será realizada considerando a implementação que realizamos no método `compareTo` da nossa classe;

```
Collections.sort(listaObj);
```

Desta forma a nossa estrutura está ordenada de acordo com as nossas pretensões e sem recurso a estruturas auxiliares;

Se o vector não é genérico, mas de um **tipo especifico** (passo 3)

```
Vector <Estudante> listaEstudantes = new Vector <Estudante>();
```

o método de comparação deve aceitar esse tipo de dados especifico e não objectos genéricos

Neste caso, a classe deve implementar a interface **`Comparable`** para esse tipo de dados (passo 1)

```
public class Estudante implements Comparable <Estudante> {…}
```

Adicionalmente, o método **`compareTo`** não deve receber um objecto genérico (**`Object`**) mas o tipo de dados especifico (passo 2)

```
public int compareTo(Estudante aEstudante) {…}
```

Exemplo de ordenação: classe **Estudante**, **ordenada pelo número**:

```
public class Estudante implements Comparable <Estudante> { // 1
     private String nome;
     private String email;
     private int    num;
     private int    telf;

     // Constructores e outros métodos

     // Ordenação através do número de aluno
     public int compareTo(Estudante aEstudante) {          // 2
          if ( num > aEstudante.getNum() )
              return 1;     // o objecto é superior ao argumento
          if ( num < aEstudante.getNum() )
              return -1;    // o objecto é inferior ao argumento
          return 0;         // os objectos são iguais
     }
     // Restante código
}
```

Exemplo de ordenação: classe **Estudante**, **ordenada pelo nome**:

```
public class Estudante implements Comparable <Estudante> { // 1
     private String nome;
     private String email;
     private int    num;
     private int    telf;

     // Constructores e outros métodos

     // Ordenação através do nome de aluno
     public int compareTo(Estudante aEstudante) {          // 2
           return nome.compareTo(aEstudante.getNome());
           // Retorna 0 de as strings forem idênticas, 1 se a string actual for
           // maior que a string em argumento, e -1 caso contrário
     }

     // Restante código
}
```

Exemplo de ordenação: lista de **`Estudante`**, representada por um ***Vector***:

```
import java.util.Collections;
import java.util.Vector;

public class VectorEstudantes {

      public static void main(String[] args) {

         Vector <Estudante> listaEstudantes = new Vector<Estudante>();   // 3

         //…

         Collections.sort(listaEstudantes);                              // 4

         //…
      }
}
```

Exemplo de ordenação: lista de **`Estudante`**, representada por um ***ArrayList***:

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

public class ArrayListEstudante {

	public static void main(String[] args) {

		ArrayList <Estudante> listaEstudantes = new ArrayList<Estudante>();

		//…

		Collections.sort(listaEstudantes);

		//…
	}
}
```

Em alternativa podemos invocar o método **`sort()`** da classe **`Arrays`**, **reescrevendo (*override*) o método `compare()`** pontualmente.

Por exemplo, para ordenar um *array* bidimensional ou matriz (**`double [][] MyTwoDimArray`**) de *doubles* considerando os valores da segunda coluna:

```
java.util.Arrays.sort(MyTwoDimArray, new java.util.Comparator<double[]>() {
public int compare(final double[] entry1, final double[] entry2) {
            final double proba1 = entry1[1]; //entry1[0] to order the first column
            final double proba2 = entry2[1]; //entry2[0] to order the first column

            if (proba1 == proba2)
                  return 0;              // elements are equal
            else
                  if (proba1 > proba2)
                        return -1;       // first element is smaller
                  else
                        return 1;        // first element is bigger
      }
});
```

O método **`compare` define como a lista será ordenada**. Manualmente indicamos qual é o maior valor. Se pretendermos a ordenação inversa, invertemos os valores 1 e -1.

Mesmo procedimento pode ser usado para **matrizes de inteiros ou *Strings***. Neste caso não necessitamos de verificar manualmente qual dos valores é maior. **A substração de inteiros devolve o valor 0 (se iguais), positivo (se o primeiro for menor) ou negativo (se o primeiro for menor)**:

```
int [][] MyTwoDimArray;
java.util.Arrays.sort(MyTwoDimArray, new Comparator<int[]>() {
      public int compare(int[] a, int[] b) {
            return b[0] - a[0];   // ordering using the first column
      }
});
```

Na manipulação de *strings* podemos recorrer ao método **`compareTo()`** **que devolve 0, 1,ou -1** caso as *strings* sejam iguais, a primeira maior ou a segunda maior, respectivamente:

```
String [][] MyTwoDimArray;
java.util.Arrays.sort(MyTwoDimArray, new Comparator<String[]>() {
      public int compare(final String[] entry1, final String[] entry2) {
            final String value1 = entry1[0]; // ordering using the first column
            final String value2 = entry2[0];
            return value1.compareTo(value2);
      }
});
```

De igual modo, para ordenar *arrays* unidimensionais:

```
int [] MyTwoDimArray;
java.util.Arrays.sort(MyTwoDimArray, new Comparator<int>() {
      public int compare(int a, int b) {
            return b - a;
      }
});

String [] MyTwoDimArray;
java.util.Arrays.sort(MyTwoDimArray, new Comparator<String>() {
      public int compare(String a, String b) {
            return a.compareTo(b);
      }
});

double [] MyTwoDimArray;
java.util.Arrays.sort(MyTwoDimArray, new Comparator<double>() {
public int compare(double a, double b) {
            if (a == b) return 0;
            else
                  if (proba1 > proba2) return -1;
                  else return 1;
      }
});
```

"***Programação Orientada a Objectos***"
António José Mendes
Departamento de Engenharia Informática, Universidade de Coimbra

"***Java in a Nutshell***", 4ª Edição, Capítulo 2 "*Java Syntax from the Ground Up*"
David Flanagan
O'Reilly, ISBN: 0596002831

"***Thinking in Java*** ", 4ª Edição, Capítulo 1 "*Introduction to Objects*"
Bruce Eckel
Prentice Hall, ISBN: 0131872486

"***The Java Tutorial – Learning the Java Language: Language Basics***"
Java Sun Microsystems
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html
http://java.sun.com/docs/books/tutorial/collections/index.html

Bibliografia complementar

Estruturas de Dados Dinâmicas

"***Fundamentos de Programação em Java 2***", Capítulo 9 "*Colecções*"
António José Mendes, Maria José Marcelino
FCA, ISBN: 9727224237

"***Java 5 e Programação por Objectos***", Capítulo 8 "*Colecções e Tipo Parametrizados*"
F. Mário Martins
FCA, ISBN: 9727225489

Referências

Tabelas

"***Java in a Nutshell***", Capítulo 2 "*Java Syntax from the Ground Up*"
"***Thinking in Java***", Capítulo 1 "*Introduction to Objects*"
"***The Java Tutorial – Learning the Java Language: Language Basics***"
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html

Vectores e Dicionários

"***Java in a Nutshell***", Capítulo 4 "*The Java Platform*"
"***Thinking in Java***", Capítulo 9 "*Collections of Objects*"
"***The Java Tutorial – Learning the Java Language: Language Basics***"
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html

Collections

"***Java in a Nutshell***", Capítulo 4 "*The Java Platform*"
"***Thinking in Java***", Capítulo 9 "*Collections of Objects*"
"***The Java Tutorial – Collections***"
http://java.sun.com/docs/books/tutorial/collections/index.html