

## **Tipos genéricos:**

Um tipo genérico ou parametrizado é um tipo que recebe como argumento outros tipos.

## **Exemplo:**

```
public class Ponto<T> {  
    private T x;  
    private T y;  
  
    public Ponto(T x , T y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

## Tipos genéricos:

```
public T getX() { return x; }
```

```
public void setX(T x) { this.x = x; }
```

```
public T getY() { return y; }
```

```
public void setY(T y) { this.y = y; }
```

**@Override**

```
public String toString() {  
    return "Ponto{" + "x=" + x + ", y=" + y + "}";  
}
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Para criar um objecto do tipo Ponto é necessário dizer ao compilador que tipo vai substituir T.

Exemplos:

```
public static void main(String[] args) {
```

```
    Ponto<Integer> i;
```

```
    Ponto<Double> d;
```

```
    i = new Ponto<Integer>(2 , 3); // autoboxing
```

```
        - conversão automática de int para Integer
```

```
    // a conversão de Integer para int designa-se por unboxing
```

```
    d = new Ponto<Double>(4.0 , 5.0);
```

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

```
System.out.println("i= " + i);  
System.out.println("d= " + d);
```

Output:

```
i= Ponto{x=2, y=3}  
d= Ponto{x=4.0, y=5.0}
```

-----

Podemos simplificar:

```
i = new Ponto<>(2 , 3);  
d = new Ponto<>(4.0 , 5.0);
```

## **Java Collections Framework (JCF)**

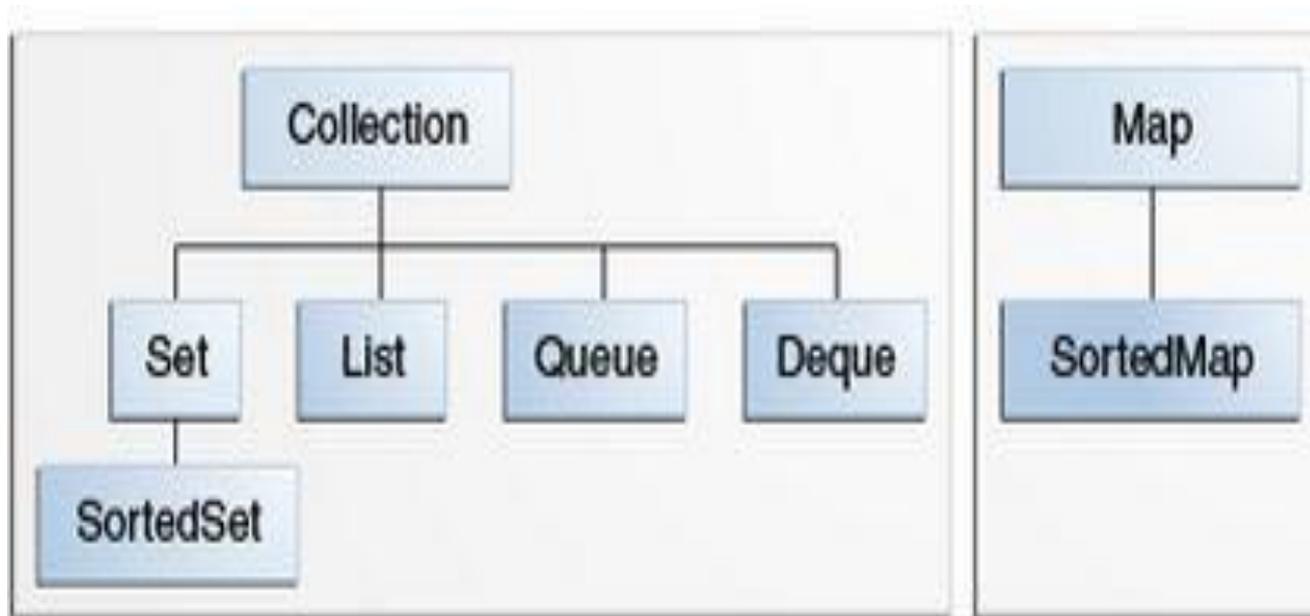
Uma coleção (collection) é um objeto que agrupa vários elementos.

A JCF consiste numa arquitetura para representação e manipulação de coleções. Contém:

- **Um conjunto de Interfaces**
- **Um conjunto de Implementações**
- **Um conjunto de algoritmos**  
**(Ex.lo: pesquisa, ordenação, ... )**

## Java Collections Framework (JCF)

### Interfaces:



**Duas hierarquias distintas.**

## **Interfaces:**

Todas as interfaces anteriores são genéricas, isto é, são declaradas como:

```
public interface Collection<E>...
```

Quando se declara uma instância de uma coleção deve-se especificar o tipo de objetos que a coleção contém.

## **Interfaces:**

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

**Set** — coleção que **não** pode conter elementos duplicados

**List** — coleção **ordenada** de elementos.

- Listas podem conter elementos duplicados;
- Os elementos de uma lista podem ser acedidos através da sua posição (um índice do tipo inteiro) ;
- Um objeto do tipo ArrayList é uma Lista.

## **Interfaces:**

**Queue** — coleção usada para a guardar elementos antes de serem processados.

- Tipicamente ordenam os elementos segundo uma ordem
- Priority queues, podem ordenar os elementos de acordo com uma ordem dada pelo utilizador.
- Numa queue (fila) FIFO todos os elementos são inseridos no final da fila.

## **Interfaces:**

### **Deque — (double ended queue)**

Coleções que podem ser usadas como FIFO (first-in, first-out) e como LIFO (last-in, first-out);

- Os novos elementos podem ser inseridos, consultados e removidos em ambas as extremidades;

## **Interfaces:**

**Map** — Correspondências ou (Maps) são coleções de objetos, parametrizadas por dois tipos. `Map<key, Value>`

Um Map é um objeto que faz corresponder (mapeia) chaves com valores;

Um Map não pode conter chaves duplicadas;

Uma chave (key) corresponde no máximo a um valor;

Uma Hashtable é um Map.

## **Interfaces:**

### **SortedSet** — versão ordenada de Set

Um sortedSet é um Set que mantém os seus elementos em ordem ascendente.

### **SortedMap** — versão ordenada de Map

Um SortedMap mantém as suas correspondências em ordem ascendente dos valores de chave.

## **A Interface Collection:**

```
public interface Collection<E> extends Iterable<E> {
```

```
//Contém métodos que executam operações básicas como:
```

```
int size() - número de elementos da coleção;
```

```
boolean isEmpty() - verifica se a coleção está vazia,
```

```
boolean contains(Object element)
```

```
- verifica se um elemento pertence à coleção.
```

```
boolean add(E element) - adiciona um elemento
```

```
boolean remove(Object element) - remove um elemento
```

```
Iterator<E> iterator() U*!*Departamento de Informática
```

## **A Interface Collection: (2)**

Contém métodos que executam operações de conjunto:

**boolean containsAll (Collection<?> coll);**

**boolean addAll(Collection<? extends E> coll );**

**boolean removeAll( Collection<?> coll);**

**boolean retainAll (Collection<?> coll);**

**void clear();**

## *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

Todos os métodos que necessitam de verificar igualdade entre objectos utilizam a igualdade baseada no método **equals**:

contains,  
add,  
Remove  
containsAll  
addAll,  
removeAll  
retainAll.

## Interface **Iterator**

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

Deve ser implementada pelas classes que pretendam iterar pelos seus elementos, um a um.

## Interface **Iterable**

```
public interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

Uma classe que implemente a interface **Iterable** oferece um **Iterator** que depois pode ser usado no ciclo for-each.

## **Percorrer uma coleção:**

### **1 . O construtor for-each**

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

//Escreve cada elemento da coleção/array na consola.

```
Ex. int [] myArray = {10, 20, 30 , 40, 50};
```

```
    for (int x: myArray)
        System.out.println( x);
```

## **Percorrer uma coleção:**

```
Ex. ArrayList<String> disciplinas = new ArrayList<String>();
    disciplinas.add("POO");
    disciplinas.add("BD");
    disciplinas.add("TC");
    disciplinas.add("ED");
    disciplinas.add("PE");

    for (String s : disciplinas)
        System.out.println(s);
```

## Percorrer uma coleção:

### 2. Iterators (2)

Um Iterator é um objeto que permite percorrer os elementos de uma coleção

O método **iterator** da interface Collection devolve um objeto do tipo Iterator

```
Iterator<E> iterator()
```

Interface Iterator:

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

## **Percorrer uma coleção:**

### **2. Iterators**

boolean hasNext() - verifica se a iteração tem mais elementos;

E next() - devolve o próximo elemento da iteração

void remove() - remove o último elemento devolvido pela operação next()

Ex.lo

```
for (Iterator<String> it = disciplinas.iterator(); it.hasNext(); )  
    System.out.println( it.next());  
}
```

**Percorrer uma coleção:**

```
for (Iterator<String> it = disciplinas.iterator(); it.hasNext();) {  
  
    String s = it.next();  
    System.out.println(s);  
    if ( s.equals("POO"))  
        it.remove();  
  
}
```

## **Implementações**

**(Classes que implementam as interfaces anteriores)**

**Implementações de uso geral (general purpose) mais usadas:**

HashSet,            implementa a interface Set

ArrayList,        implementa a interface List

HashMap,         implementa a interface Map

LinkedList,       implementa a interface Queue

ArrayDeque,      implementa a interface Deque

## **Algoritmos:**

**(A classe Collections contém algumas funções de classe (static) que operam em coleções)**

### **1 - Sorting**

. void **sort** ( List<T> list) \*

\*Os elementos da lista têm de implementar a interface Comparable

. void **sort** (List<T> list, Comparator <? super T> c )

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Ex.10**

```
Collections.sort(disciplinas);
```

```
for (Iterator<String> it = disciplinas.iterator(); it.hasNext();)  
    System.out.println( it.next());
```

```
Ou (for (String s: disciplinas )  
    System.out.println(s); )
```

```
Output:      BD  
            ED  
            PE  
            POO  
            TC
```

## **2 - Shuffling**

**(Baralha os elementos usando um gerador de valores aleatórios)**

. void **shuffle** (List<?> list)

. void **shuffle** (List<?>List , Random rnd)

# *Programação Orientada a Objectos - P. Prata, P. Fazendeiro*

## **Ex.lo**

```
System.out.println ("Shuffling");  
Collections.shuffle(disciplinas);
```

```
for (String s: disciplinas )  
    System.out.println(s); )
```

Um output:

Shuffling

PE

ED

BD

TC

POO

### 3- Manipulação de dados

. **Reverse** (inverte a ordem dos elementos)

**reverse (List<?> list);**

Ex.lo

`Collections.reverse(disciplinas);`

. **Fill** (substitui cada elemento por um valor dado)

**fill (List<? super T> list, T obj)**

Ex.lo

`Collections.fill(disciplinas, "ops");`

### **3- Manipulação de dados**

- . **copy** - copia os valores de uma lista para outra
- . **swap** - troca dois elementos dadas as suas posições
- . **addAll** – adiciona um conjunto de elementos a uma lista

## 4 - Pesquisa

. binarySearch

Pesquisa um valor (key) numa lista ordenada.

Collections.sort(disciplinas);

**int pos = Collections.binarySearch(disciplinas, "XPTO");**

Se a lista contém o valor, devolve a sua posição, caso contrário, devolve o valor “position” tal que  $-(\text{position}) - 1$  é o ponto onde deve ser inserido o elemento na lista, isto é; o índice do primeiro elemento maior do que o valor pesquisado ou o `list.size()` se todos os valores da lista forem menores que o valor pesquisado.

## 4 - Pesquisa

```
Collections.sort(disciplinas);  
for (String s: disciplinas )  
    System.out.println(s);
```

BD

ED

PE

POO

TC

```
pos= Collections.binarySearch(disciplinas, "XPTO");  
    System.out.println("pos: "+ pos);
```

**-6**

```
pos= Collections.binarySearch(disciplinas, "ALGA");  
    System.out.println("pos: "+ pos);
```

**-1**

## 4 - Pesquisa

```
pos= Collections.binarySearch(disciplinas, "PE");  
    System.out.println("pos: "+ pos);
```

**2**

```
pos= Collections.binarySearch(disciplinas, "IHC");  
    System.out.println("pos: "+ pos);
```

**-3**

## 4 - Pesquisa

```
int pos = Collections.binarySearch(list, key);  
    if (pos < 0)  
        list.add(-pos-1, key);
```

- Pesquisa o valor (key) na lista e, caso não esteja na lista, insere.

## 5 – Máximo e Mínimo

**String s**

**s = Collections.max(disciplinas);**

**s = Collections.min(disciplinas);**

...

## **Classe Arrays**

Explorar ...