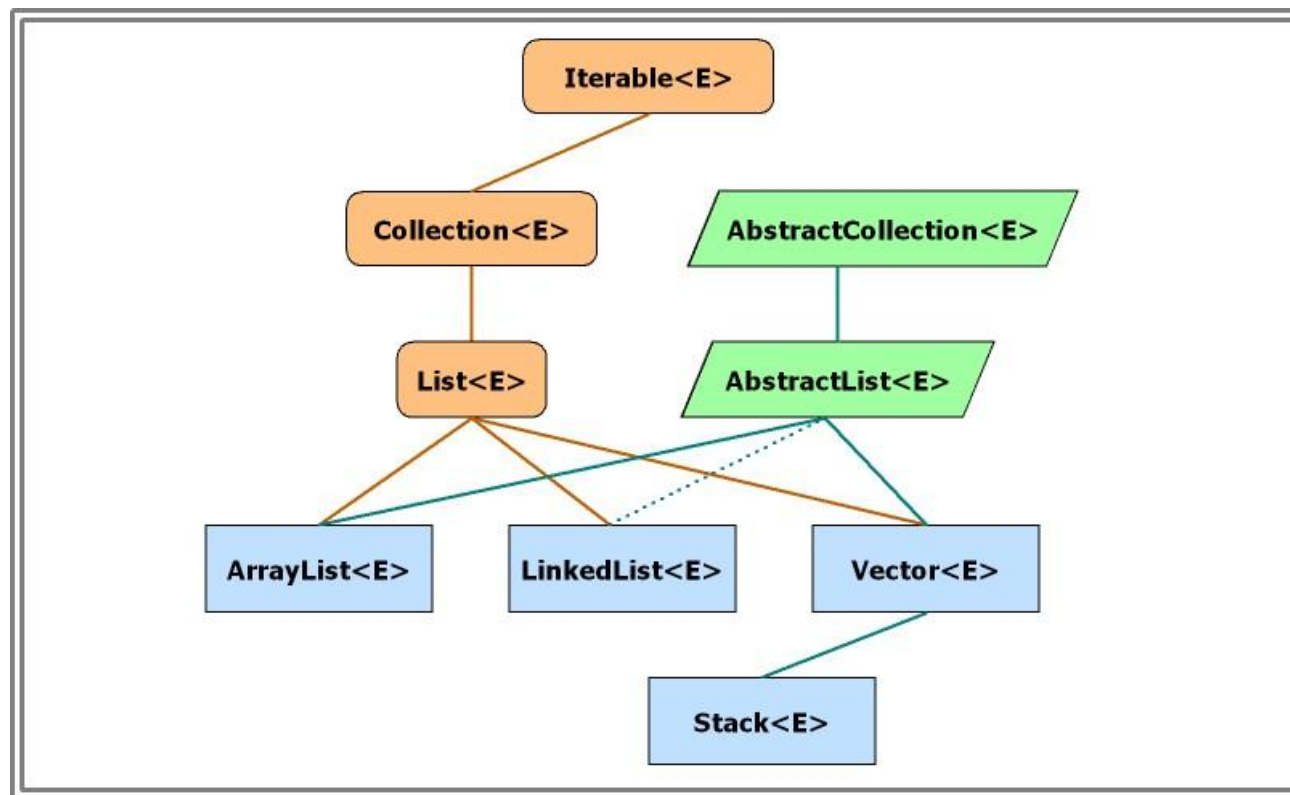


# INTERFACES DE JAVA

O JCF É UMA ARQUITECTURA QUE SE BASEIA EM TRÊS ENTIDADES FUNDAMENTAIS:

- A) INTERFACES;
- B) CLASSES ABSTRACTAS;
- C) CLASSES CONCRETAS – IMPLEMENTAÇÕES;



**INTERFACES** SÃO ESPECIFICAÇÕES **SINTÁCTICAS** (ALGUMAS GENÉRICAS IE. PARAMETRIZADAS) DE CONSTANTES E MÉTODOS QUE, EM GERAL, CORRESPONDEM A **PROPRIEDADES** QUE QUALQUER CLASSE QUE AS PRETENDA TER/SATISFAZER PODE IMPLEMENTAR, DESDE QUE **FORNEÇA IMPLEMENTAÇÕES** PARA TODOS OS MÉTODOS ESPECIFICADOS NA INTERFACE.

```
public interface Colorivel {  
    public abstract int getCor();  
    public abstract void setCor(int cor);  
}
```

```
public interface Datavel {  
    void setData(GregorianCalendar data);  
    GregorianCalendar getData();  
}
```

**ALGUMAS INTERFACES DE JAVA NEM SEQUER DEFINEM COMPORTAMENTO, SÃO APENAS ETIQUETAS PARA O COMPILADOR/INTERPRETADOR (“**MARKER INTERFACES**”):**

```
public interface Serializable { }  
/** AS CLASSES QUE DECLARAREM IMPLEMENTAR ESTA INTERFACE PODEM SER GRAVADAS EM  
FICHEIROS DE OBJECTOS, ObjectStreams */
```

```
public interface Cloneable { }  
/** CLASSES QUE USAM O MÉTODO CLONE() DE OBJECT */ NOTA: NÃO USAMOS !!
```

**INTERFACES** SÃO **TIPOS**, TAL COMO AS CLASSES, SENDO COMPATÍVEIS COM O TIPO DAS CLASSES QUE AS IMPLEMENTAM, CF.

```
List<String> nomes = new ArrayList<String>();  
                = new LinkedList<Ponto2D>();
```

**INTERFACES** SÃO ESPECIFICAÇÕES IMPLEMENTADAS PELAS **CLASSES**.

```
public class PontoComCor implements Colorivel {  
    // variáveis de instância  
    private int x, y;  
    // variáveis para implementar Colorivel  
    private int cor = 0;  
    // métodos de instância  
    .....  
    // implementação de Colorivel  
    public int getCor() { return cor; };  
    public void setCor(int ncor) { cor = ncor; };  
}
```

**ASSIM:**

```
PontoComCor pc = new PontoComCor();  
Colorivel col = new PontoComCor(10.0, 5.0);  
  
x = pc.getX(); cor = pc.getCor(); cor = col.getCor(); col.setCor(99);  
ERRADAS: col.getX(); col.somaPonto(1.0, 5.0);    /** cf. as suas API */
```

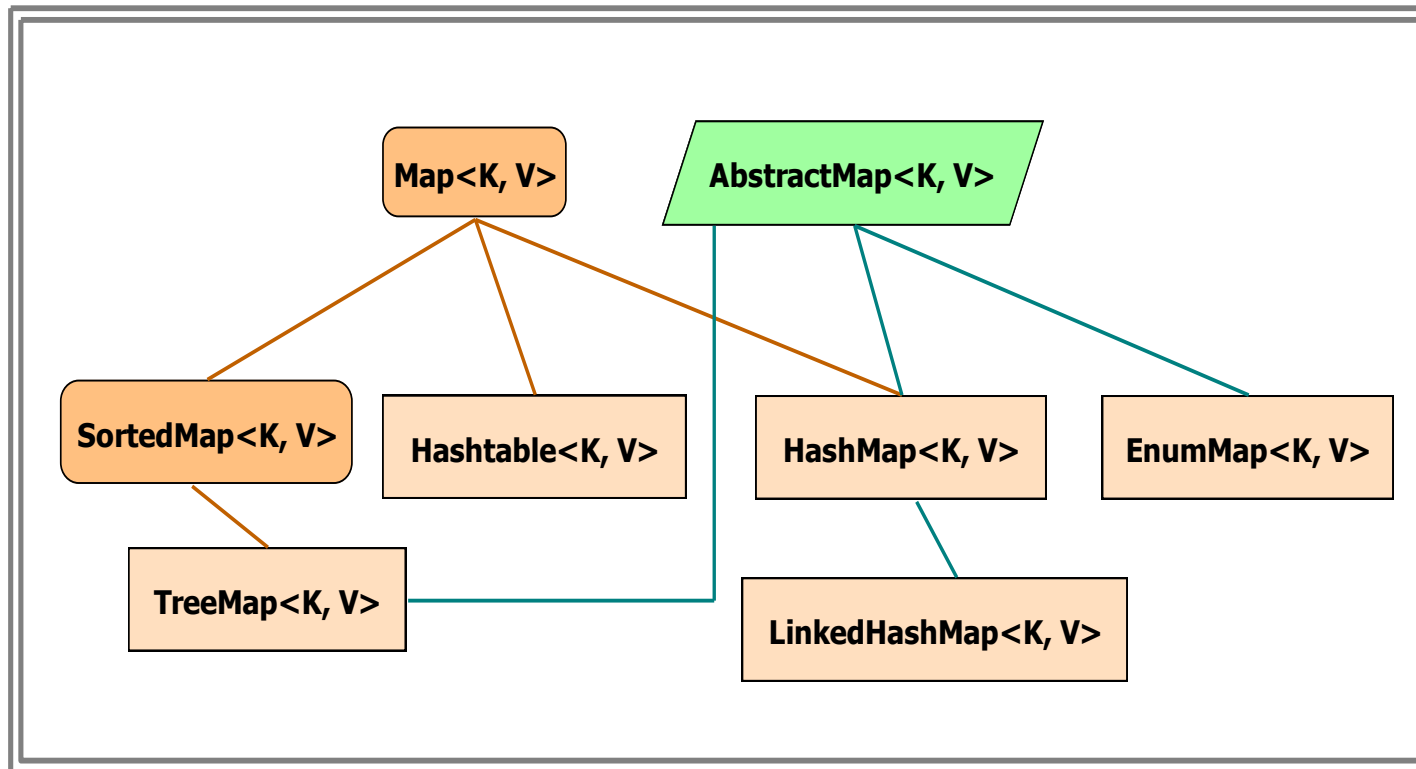
## UMA CLASSE PODE IMPLEMENTAR VÁRIAS INTERFACES:

```
public class PontoComCorDatavel implements Colorivel, Datavel {
    // variáveis de instância
    private int x, y;
    // variáveis para implementar Colorível
    private int cor = 0;
    // variáveis para implementar Datavel
    private GregorianCalendar data = null;

    // métodos de instância
    .....
    // implementação de Colorivel
    public int getCor() { return cor; };
    public void setCor(int ncor) { cor = ncor; };

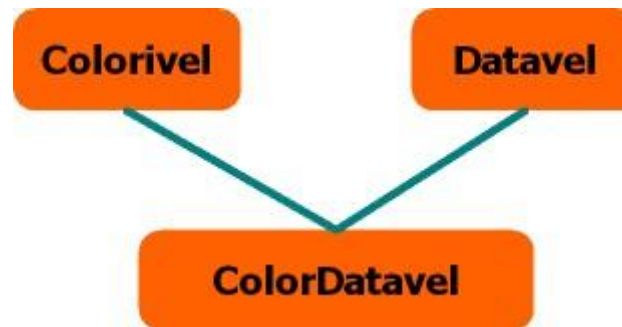
    // implementação de Datavel
    public void setData(GregorianCalendar data) {
        this.data = (GregorianCalendar) data.clone();
    }
    public GregorianCalendar getData() {
        return (GregorianCalendar) data.clone();
    }
}
```

AS **INTERFACES** DE JAVA PODEM FAZER PARTE DE UMA **HIERARQUIA DE INTERFACES**. MAS PODEM SER INTERFACES LIVRES, DESLIGADAS DE TODAS AS OUTRAS. POR VEZES DÁ MESMO JEITO COLOCAR CERTAS INTERFACES NUMA HIERARQUIA MÚLTIPLA. POR EXEMPLO, EM JCF:



ASSIM, A CLASSE **TreeMap<K, V>** IMPLEMENTA A INTERFACE **SortedMap<K, V>** O QUE SIGNIFICA QUE POR HERANÇA, IMPLEMENTA A INTERFACE **Map<K, V>** MAIS OS MÉTODOS ADICIONAIS ESPECIFICADOS EM **SortedMap<K, V>**.

NO NOSSO EXEMPLO ANTERIOR, PODERÍAMOS TER:



E ENTÃO A CLASSE `PontoComCorDatavel` PODERIA SER DECLARADA APENAS COMO:

```
public class PontoComCorDatavel implements ColorDatavel {
```

---

ALGUMAS INTERFACES DE JAVA SÃO MUITO IMPORTANTES. POR EXEMPLO A INTERFACE `Comparable<T>` DEFINIDA COMO:

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```

QUALQUER CLASSE QUE PRETENDA TER UMA **ORDEM NATURAL** (>, = <) DEFINIDA PARA AS SUAS INSTÂNCIAS **DEVE IMPLEMENTAR ESTA INTERFACE**. A CLASSE `String` E AS CLASSES “**WRAPPER**” IMPLEMENTAM ESTA INTERFACE `Comparable<T>` E POR ISSO SÃO AUTOMATICAMENTE ORDENADAS EM `TreeSet<E>` E `TreeMap<K,V>`.

**PORÉM, O QUE FAZER SE SE PRETENDER ORDENAR OBJECTOS FORA DA SUA ORDEM NATURAL OU SE SE PRETENDER ORDENAR OBJECTOS QUE NÃO IMPLEMENTAM `Comparable<T>`?**

**USO DA INTERFACE `Comparator<E>` EM `TreeSet<E>` (`E` `TreeMap<K,V>` para `K`)**

AS COLECÇÕES IMPLEMENTADAS USANDO ÁRVORES POSSUEM UM CONSTRUTOR QUE PERMITE QUE SE DEFINA UM ALGORITMO DE COMPARAÇÃO A USAR NA ORDENAÇÃO DOS OBJECTOS. NO CASO, POR EXEMPLO, DE `TreeSet<E>`,

```
new TreeSet<E>(Comparator<E>)
```

O PARÂMETRO `Comparator<E>` É UMA INTERFACE PARAMETRIZADA DE JAVA, QUE ESTÁ DEFINIDA COMO:

```
public interface Comparator<T> {  
    int compare(T obj1, T obj2); // 0, 1 ou -1  
}
```

ASSIM, SE, POR EXEMPLO, PRETENDEREMOS CRIAR UM `TreeSet<Ponto2D>` DE FORMA A QUE OS PONTOS ESTEJAM ORDENADOS DE ALGUM MODO, TEMOS QUE CRIAR UMA CLASSE QUE **IMPLEMENTE** A INTERFACE `Comparator<Ponto2D>`, OU SEJA, IMPLEMENTE O MÉTODO

```
int compare(Ponto2D obj1, Ponto2D obj2)
```

VEJAMOS COMO PODERIA SER TAL CLASSE (1 DE MUITAS HIPÓTESES):

```
import java.util.Comparator;
public class PontoComparator implements Comparator<Ponto2D> {
    public int compare(Ponto2D p1, Ponto2D p2) {
        if(p1.getX() < p2.getX()) return -1;
        if(p1.getX() > p2.getX()) return 1;
        if(p1.getY() < p2.getY()) return -1;
        if(p1.getY() > p2.getY()) return 1; else return 0;
    }
}
```

TEMOS POIS UMA CLASSE QUE IMPLEMENTA A INTERFACE `Comparator<Ponto2D>`.  
VAMOS AGORA TESTÁ-LA, CRIANDO UM `TreeSet<Ponto2D>` USANDO O CONSTRUTOR NA  
FORMA

```
new TreeSet<Ponto2D>(new PontoComparator())
```

E ESCRREVENDO UM PROGRAMA DE TESTE QUE VAI VERIFICAR SE OS ELEMENTOS DO  
CONJUNTO SÃO DE FACTO ENUMERADOS CONFORME O ALGORITMO DE ORDENAÇÃO  
DEFINIDO.



```

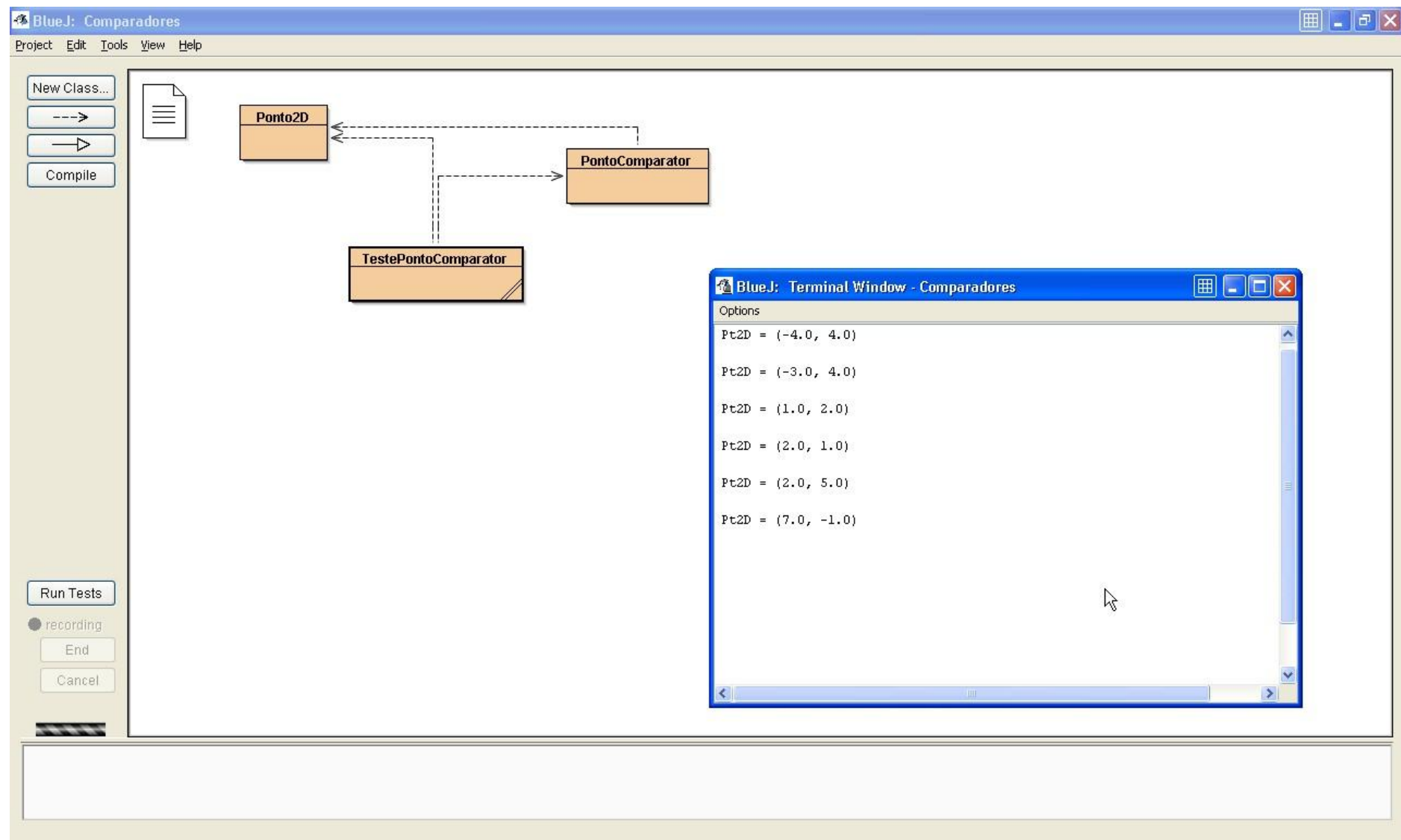
/**
 * TestePontoComparator
 *
 * @author F. Mário Martins
 * @version 1/2006
 */
import static java.lang.System.out;
import java.util.*;
public class TestePontoComparator {

    public static void main(String args[]) {
        TreeSet<Ponto2D> plano =
            new TreeSet<Ponto2D>(new PontoComparator());
        plano.add( new Ponto2D(7.0, -1.0) );
        plano.add( new Ponto2D(1.0, 2.0) );
        plano.add( new Ponto2D(-3.0, 4.0) );
        plano.add( new Ponto2D(2.0, 5.0) );
        plano.add( new Ponto2D(-4.0, 4.0) );
        plano.add( new Ponto2D(2.0, 1.0) );

        for(Ponto2D p : plano) out.println(p.toString());
    }
}

```

## RESULTADOS:



RELEMBRANDO A CLASSE **Plano**, VAMOS REDEFINI-LA AGORA COMO,

```
public class Plano implements Serializable {
```

```
    private TreeSet<Ponto2D> pontos;
```

```
    // Construtor por omissão: usa o comparador PontoComparator
```

```
    public Plano() { pontos = new TreeSet<Ponto2D>( new PontoComparator() ); }
```

```
    // Recebe um conjunto de pontos e um algoritmo de ordenação e copia os
```

```
    // pontos para a variável de instância, ordenando-os por tal ordem
```

```
    public Plano(Set<Ponto2D> pts, Comparator<Ponto2D> comp) {
```

```
        pontos = new TreeSet<Ponto2D>( comp );
```

```
        for(Ponto2D p : pts) pontos.add(p.clone());
```

```
    }
```

```
    // Métodos de instância
```

```
    ....
```

```
    // Método que devolve uma cópia do TreeSet<Ponto2D>
```

```
    // ordenado pelo algoritmo de ordenação passado como parâmetro
```

```
    public TreeSet<Ponto2D> ordenaPor(Comparator<Ponto2D> comp) {
```

```
        TreeSet<Ponto2D> aux = new TreeSet<Ponto2D>(comp);
```

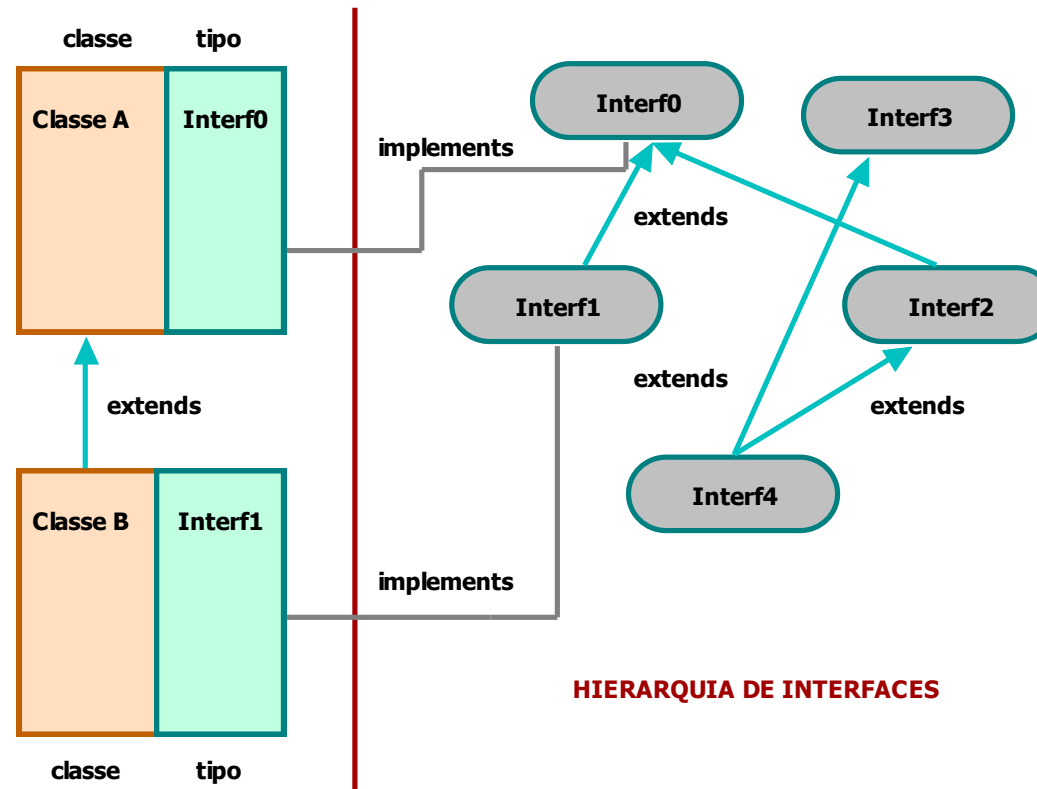
```
        for(Ponto2D p : pontos) aux.add(p.clone());
```

```
        return aux;
```

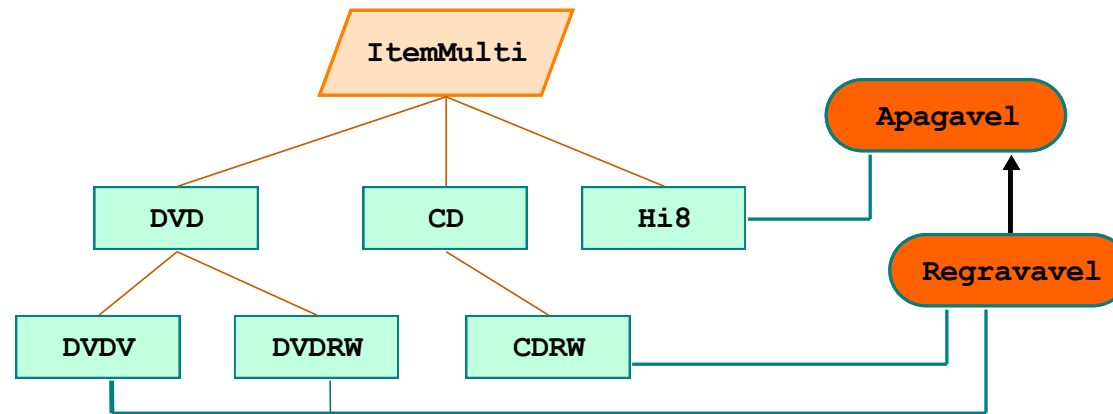
```
    }
```

```
}
```

**CLASSES E INTERFACES JAVA PERTENCEM A DUAS HIERARQUIAS DISTINTAS. A HIERARQUIA DE CLASSES É DE TIPO SIMPLES ENQUANTO QUE A DE CLASSES É UMA HIERARQUIA MÚLTIPLA (1 INTERFACE PODE HERDAR DE MAIS DO QUE UMA).**



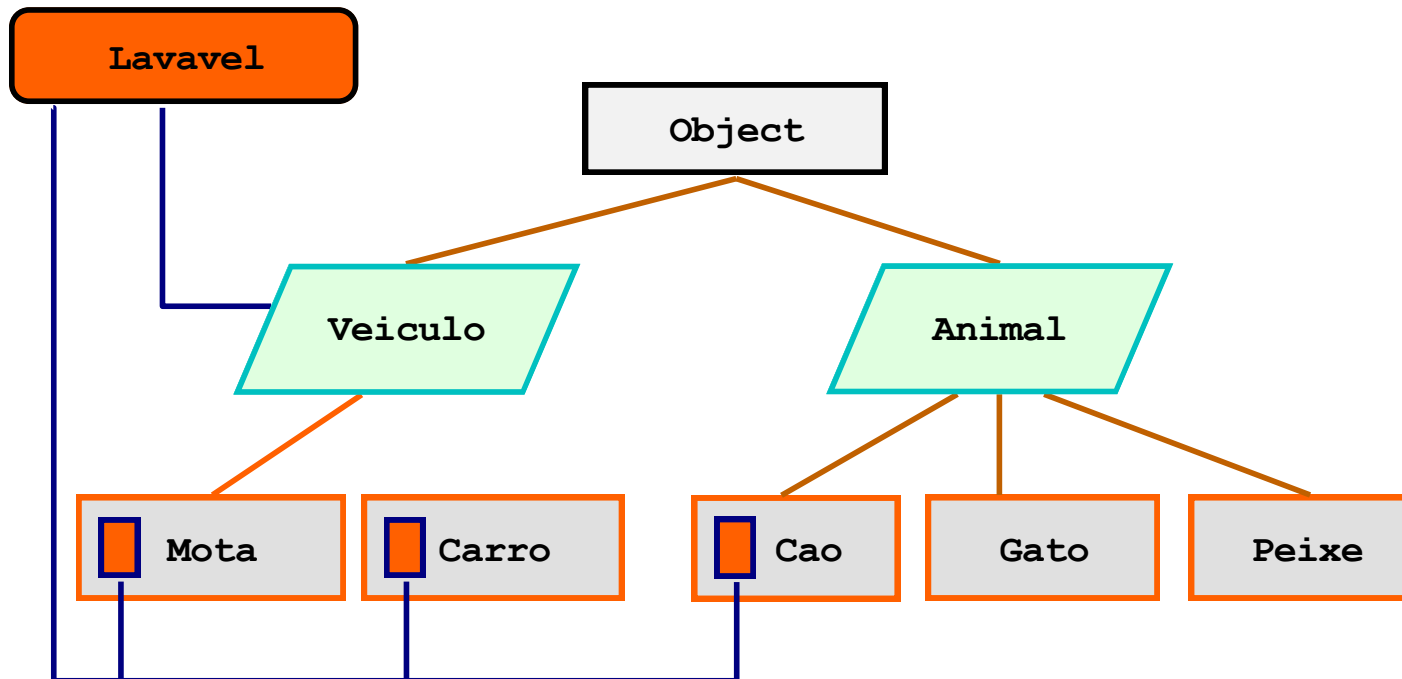
## EXEMPLOS:



```
public interface Apagavel {  
    public abstract void apaga();  
}  
  
public interface Regravavel extends Apagavel {  
    public abstract void grava(byte[] cont);  
}
```

## INTERFACES PODEM SER IMPLEMENTADAS POR QUAISQUER CLASSES EM QUALQUER PONTO DA HIERARQUIA

```
public interface Lavavel {  
    public void aLavar(int tempo);  
}
```



## INTERFACES NÃO INTERFEREM NA HIERARQUIA DE CLASSES

## INTERFACES VERSUS CLASSES ABSTRACTAS

A TABELA APRESENTA MAIS ALGUMAS DIFERENÇAS ENTRE A UTILIZAÇÃO DE INTERFACES E DE CLASSES ABSTRACTAS, RELATIVAMENTE A PROPRIEDADES DIVERSAS, NÃO TANTO CONCEPTUAIS MAS MAIS RELACIONADAS COM A FUNCIONALIDADE E RESTRIÇÕES NA UTILIZAÇÃO.

Propriedades	Interfaces	Classes Abstractas
<b>Herança Múltipla</b>	Uma classe pode implementar várias interfaces	Uma classe só pode "estender" uma classe abstracta
<b>Implementação</b>	Não aceita nenhum código	O código que se pretender, cf. as necessidades da concepção
<b>Constantes</b>	Apenas <code>static final</code> . Podem ser usadas pelas classes implementadoras sem qualificadores	De classe ou de instância e código de inicialização se necessário
<b>Relação <i>is-a</i></b>	Em geral não descrevem a identidade principal de uma classe, mas capacidades periféricas aplicáveis a outras classes	Descreve uma identidade básica de todos os seus descendentes
<b>Adicionar funcionalidade</b>	Não deve ser feito pois implicaria refazer todas as implementações da interface. Usar antes sub-interfaces	Sem problema para as subclasses desde que o método seja codificado na classe abstracta
<b>Homogeneidade</b>	Classes que as implementam são homogéneas na API	Subclasses homogéneas em API e que partilham algum código, por exemplo, classes de dados
<b>Velocidade (não é relevante)</b>	Requer extra indirectão na procura	Procura directa do método

Interfaces *versus* classes abstractas