

# Prácticas de laboratorio de LTP (Parte I : Java)

## Práctica 3: Interfaces



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA

Jose Luis Pérez  
[jlperez@dsic.upv.es](mailto:jlperez@dsic.upv.es)

# Introducción : Interfaces en Java

- Podríamos decir que una interfaz es un tipo especial de clase que es una **clase totalmente abstracta** (donde todos sus métodos son abstractos).
- Las interfaces introducen cierta flexibilidad en la herencia de Java, y con ello **incrementan la capacidad del polimorfismo en el lenguaje**.
- Las interfaces son clases que se usan para especificar **TAD's** (Tipos Abstractos de Datos). Al implementar una interfaz, están obligadas ellas y/o sus derivadas a implementar los métodos abstractos heredados de la interfaz.
- El uso de TAD's da lugar a **programas mas robustos y menos propensos a errores**.
- Al definir interfaces también se permite la existencia de **variables polimórficas** definiéndolas con el tipo de una interfaz. Lo que permite asignar a estas variables cualquier instancia de una clase heredada de ellas.

# Interfaces en Java: Sintaxis

Dadas la relaciones de herencia entre las clases e interfaces que se muestra en la figura, se proponen posibles definiciones para las mismas:

```
public interface A<T> { ... }
```

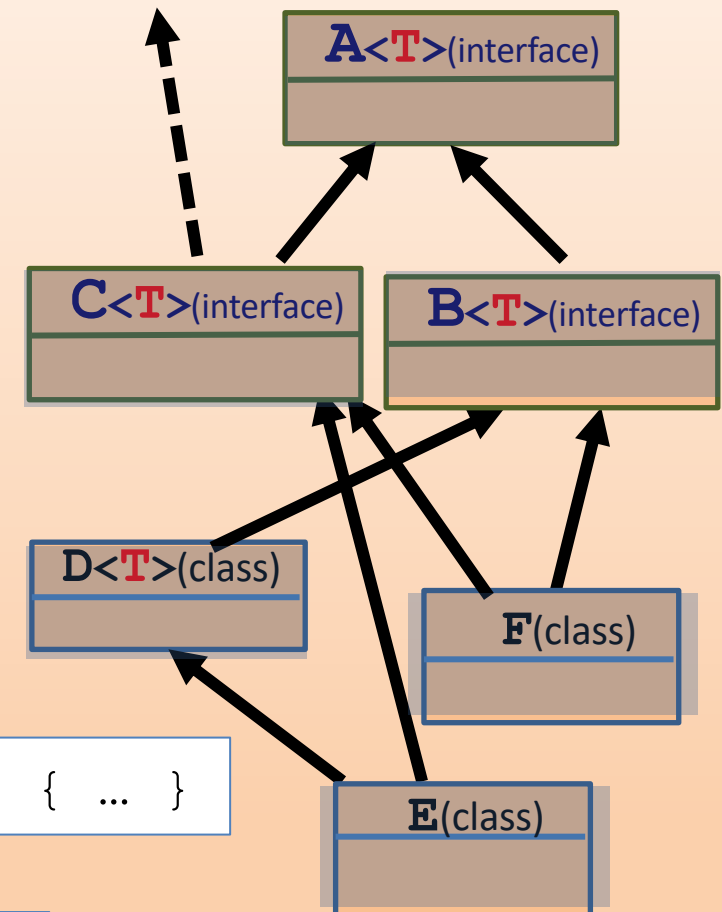
```
public interface B<T> extends A<T> { ... }
```

```
public interface C<T> extends A<T>, ... { ... }
```

```
public class D<T> implements B<T> { ... }
```

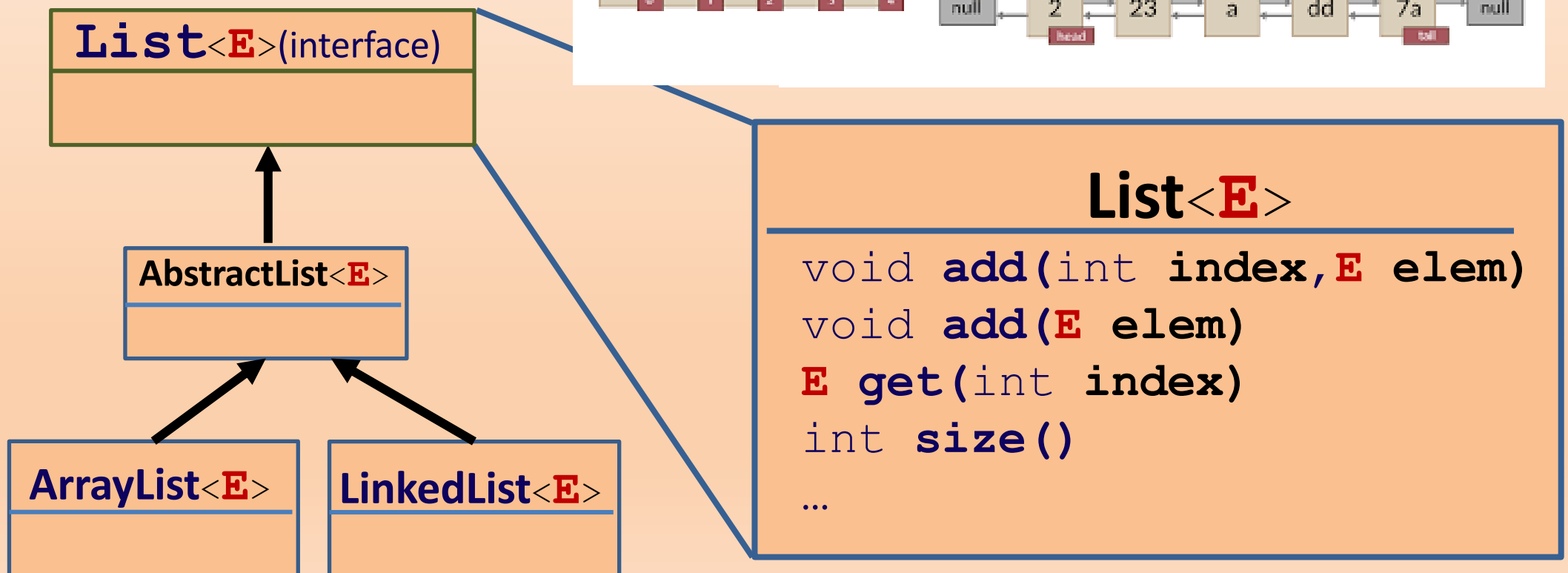
```
public class E extends D<E> implements C<E> { ... }
```

```
public class F implements C<F>, B<F> { ... }
```



# Ejemplo de TAD's: La interfaz List

El API de Java, dentro del paquete `java.útil`, está definida la interfaz `List` que describe el las operaciones que debe implementar una clase para poder ver sus elementos como una lista:



# Variables polimórficas de tipo **List**: Ejemplo

En el siguiente ejemplo se crean **variables polimórficas** referenciándolas con el tipo de la interfaz **List**:

```
public static void pruebaL() {  
    String[] carsV = {"Volvo", "BMW", "Ford"};  
    List<String> carsL = convertToList(carsV);  
    System.out.println("lista marcas de coches: \n"+carsL);  
  
    Figure[] FiguresV = {new Circle(2,3,5),  
                        new Triangle(2,3,5,6), new Rectangle(2,3,5,8)};  
    List<Figure> figuresL = convertToList(FiguresV);  
    System.out.println("lista de Figuras: \n"+figuresL);  
}
```

```
public static <T> List<T> convertToList (T[] theArray) {  
    int i;  
    List<T> l = new ArrayList<T> ();  
    for (i=0; i<theArray.length; i++)  
        l.add(theArray[i]);  
    return l;  
}
```

El método genérico **convertToList** convierte una array en una lista.

La implementación de la lista con **ArrayList** puede ser cambiada por **LinkedList** sin modificar el resto del código.

# 1. Objetivo / Planteamiento:

En esta tercera práctica se plantea la ampliación, utilizando interfaces, de la solución que se realizó en la **práctica 1**.

En los ejercicios de esta práctica, se añadirán funcionalidades adicionales a las clases que implementaste entonces, en 3 pasos:

a) Usando interfaces predefinidas

b) Extendiendo una interfaz predefinida

c) Diseñando una nueva interfaz

**Nota:** en el fichero “**testing.java**” de Poliformat, está definida la clase **testing**, que contiene métodos para probar todos los ejercicios de las **prácticas 1 y 3**.

# Añadir un nuevo paquete en el proyecto **ltp**

**Ejercicio 1:** En el proyecto **ltp**, crea un paquete de nombre **practica3**. Añade a este paquete una copia de todas las clases implementadas en la primera práctica (disponibles en la carpeta **ltp/practica1**). Así, podrás resolver los ejercicios siguientes sobre las clases del paquete **practica3**, sin modificar ni perder nada de lo desarrollado en el paquete **practica1**.

La forma más sencilla de realizar este ejercicio es crear una carpeta **practica3** dentro de la carpeta **ltp**, y copiar en ella todos los ficheros **.java**, ya creados, del directorio **practica1**.

Abrir dicho proyecto y utilizando la opción “**New Package**” seleccionar el directorio **practica3**.

The screenshot shows the BlueJ IDE interface. On the left, the 'Edit' menu is open, and the 'New Package...' option (with a keyboard shortcut of Ctrl+R) is highlighted. A red arrow points from this menu item to the 'practica3' package icon in the project view on the right. The project view shows a hierarchy with 'ltp' as the root, containing 'practica1', 'practica2', and 'practica3'. The 'practica3' package is highlighted with a red box. Another red arrow points from the 'practica3' package icon to a code editor window on the right. The code editor shows the following Java code:

```
package practica3;

public abstract class Figure {
    private double x;
    private double y;
}
```

A red arrow points from the 'package practica3;' line to the text 'Comprobar que el nombre del paquete en las clases sea practica3'.

Aparece el nuevo paquete **practica3**

Comprobar que el nombre del paquete en las clases sea **practica3**

## 2. Uso de una interfaz predefinida:

### 2.2. La interfaz Comparable


La interfaz **Comparable** es de uso común para comparar objetos de una misma clase entre sí. Esta clase solo define un método, **compareTo**:

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

La norma de uso de este método establece cual es el resultado de aplicar este método a un objeto **e1** recibiendo como parámetro otro objeto **e2**. Veamos un ejemplo:

```
double e1 = 4.1;  
double e2 = 5.3;  
if (e1.compareTo(e2) > 0) {  
    System.out.println("e1 es mayor que e2"); }  
if (e1.compareTo(e2) < 0) {  
    System.out.println("e1 es menor que e2"); }  
if (e1.compareTo(e2) == 0) {  
    System.out.println("e1 es igual que e2"); }
```

Puesto que **e1** es menor que **e2** el resultado de **compareTo** debe ser negativo.





## 2.2. La interfaz Comparable : Ejemplo

Esta norma puede concretarse para cada clase que la implemente, y permite establecer un orden entre los elementos de dicha clase, o cambiar el orden natural que ya pueda estar definido.

En este ejemplo modificamos el ejemplo anterior **pruebaL** para que muestre las listas ordenadas utilizando **Sort**. vemos que la clase **String**, ya tiene un orden natural entre sus elementos mientras que la clase **Figure** no lo tiene:

```
public static void pruebaLSort() {  
    String[] carsV = {"Volvo", "BMW", "Ford"};  
    List<String> carsL = convertToList(carsV);  
    Collections.sort(carsL); COMPILA CORRECTAMENTE  
    System.out.println("lista de marcas de coche: \n"+carsL);  
  
    Figure[] FiguresV = {new Circle(2,3,5),  
                        new Triangle(2,3,5,6), new Rectangle(2,3,5,8)};  
    List<Figure> figuresL = convertToList(FiguresV);  
    Collections.sort(figuresL); ERROR COMPILACIÓN  
    System.out.println("lista de Figuras: \n"+figuresL);  
}
```

La clase **String** implementa **Comparable**, y utiliza **compareTo** para ordenar los elementos de **carsL**.

Porque no hay un orden definido para las figuras

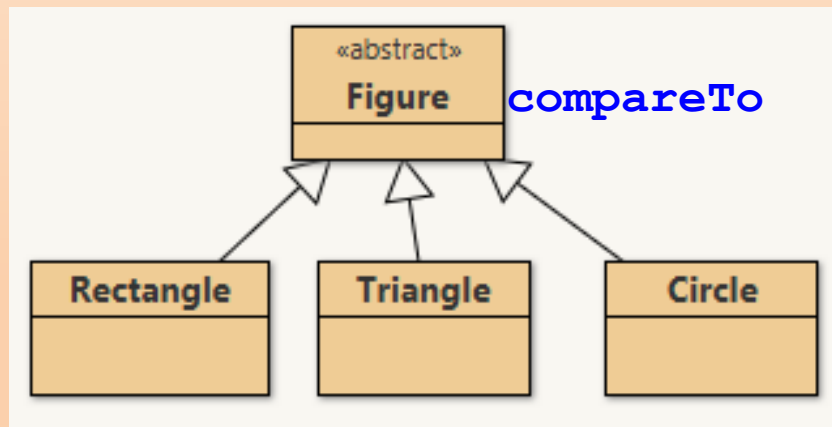
# Implementar Comparable en clase Figure

**Ejercicio 2:** Realiza los cambios necesarios en la clase **Figure** de forma que pueda determinarse cuando una figura es mayor (más grande) que otra. Esta clase debe implementar el interfaz **Comparable** para que todos los objetos de sus clases derivadas sean comparables entre sí.

**NOTA:** En esta implementación, el tipo genérico **T** de la interfaz debe concretarse en el tipo de una clase (en este caso, **Figure**).

## Opción CORRECTA

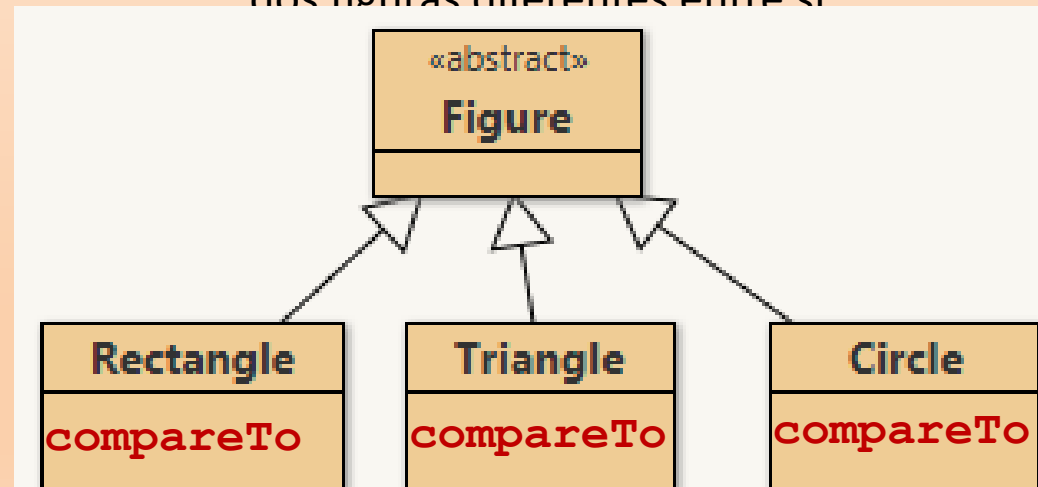
implementar **compareTo** en **Figure**, ya que se pueden comparar dos figuras diferentes entre sí



```
Circle c1 = new Circle(1.0, 6.0, 6.0);
Circle c2 = new Circle(1.0, 6.0, 8.0);
System.out.println(c1.compareTo(c2));
```

## Opción INCORRECTA

implementar **compareTo** en los herederos de **Figure**, ya que **NO** se pueden comparar dos figuras diferentes entre sí



Puesto que el área **c1** es menor que la de **c2** el resultado debe ser negativo.

## 2.2. La interfaz Comparable : Ejemplo

Después de implementar **Comparable** en la clase **Figure** (ver **ejercicio 2**), se puede ejecutar el método **pruebaLSort** sin errores:

```
public static void pruebaLSort() {
    String[] carsV = {"Volvo", "BMW", "Ford"};
    List<String> carsL = convertToList(carsV);
    Collections.sort(carsL);
    System.out.println("lista marcas de coches: \n"+carsL);

    Figure[] FiguresV = {new Circle(2, 3, 5),
                        new Triangle(2, 3, 5, 6), new Rectangle(2, 3, 5, 8)};
    List<Figure> figuresL = convertToList(FiguresV);
    Collections.sort(figuresL); COMPILA CORRECTAMENTE
    System.out.println("lista de Figuras: \n"+figuresL);
}

public static <T> List<T> convertToList (T[] theArray) {
    int i;
    List<T> l = new ArrayList<T>();
    for (i=0; i<theArray.length; i++) {
        l.add(theArray[i]);
    }
    return l;
}
```

## 2.3. La interfaz List : Ejemplo

En el siguiente ejercicio se plantea crear un método en **FiguresGroup** que devuelva una lista de figuras ordenada.

La implementación de este método permitirá, a **FiguresGroup**, utilizar la operaciones de **List<E>** sin tener que implementar la interfaz.

```
public class FiguresGroup {  
    private static final int NUM_FIGURES = 10;  
    private Figure[] figuresList = new Figure[NUM_FIGURES];  
    private int numF = 0;
```

Recordamos que las figuras están almacenadas en el atributo **figuresList**, de tipo **array**

Se añaden los elementos del grupo a la lista de uno en uno

```
    public void add(Figure f) { figuresList[numF++] = f; }
```

```
    public List<Figure> orderedListSort () {  
        List<Figure> l = new ArrayList<Figure> ();  
        for (int i = 0; i < numF; i++) {  
            l.add(figuresList[i]);  
        }  
        Collections.sort(l);  
        return l;  
    }
```

El método podría ser invocado desde **FiguresGroupUse** de esta forma

```
    public static void main(String[] args) {  
        FiguresGroup g = new FiguresGroup();  
        ... // Añadir elementos al grupo  
        System.out.println(g.orderedListSort());  
    }
```

# Algoritmo de inserción directa: `orderedList`

**Ejercicio 3:** Implementa el método `orderedList` en la clase `FiguresGroup` usando las operaciones de la interfaz `List` especificadas anteriormente y el comparador de la clase `Comparable` (método `compareTo`):

```
public List<Figure> orderedList()
```

Con el fin poner en práctica el API de `List<E>` utilizaremos el algoritmo de **inserción directa**. El resultado del método `orderedList` debe ser el mismo que el retornado por el método `orderedListShort`, y podemos utilizarlo para comprobar el resultado.

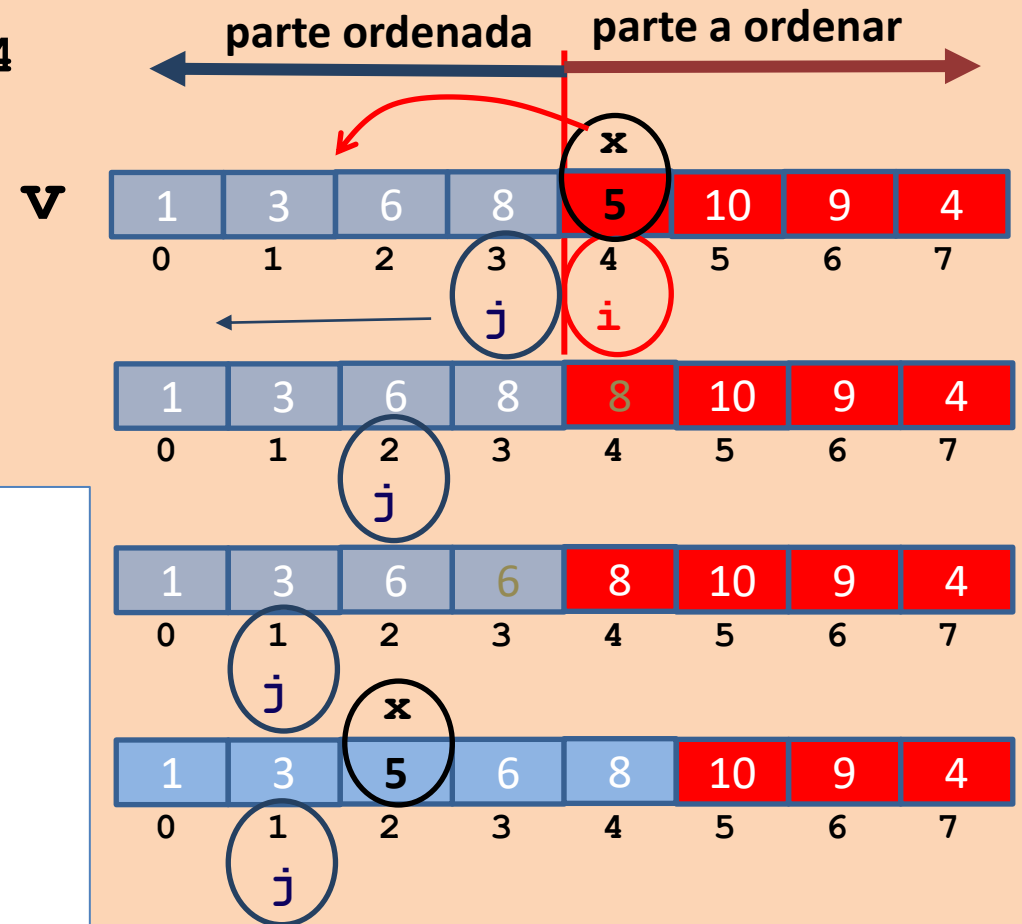
El método `orderedList` retornará una lista de figuras ordenada. Podemos probar el método con las siguiente instrucciones en `FiguresGroupUse`:

```
public static void main(String[] args) {  
    FiguresGroup g = new FiguresGroup();  
    g.add(new Circle(1.0, 6.0, 6.0));  
    g.add(new Rectangle(2.0, 5.0, 10.0, 12.0));  
    g.add(new Triangle(3.0, 4.0, 10.0, 2.0));  
    g.add(new Circle(4.0, 3.0, 1.0));  
    g.add(new Triangle(5.0, 1.0, 1.0, 2.0));  
    g.add(new Square(6.0, 7.0, 15));  
    g.add(new Rectangle(7.0, 2.0, 1.0, 3.0));  
    System.out.println(g.orderedList());  
}
```

# Algoritmo de inserción directa: `insDirecta (Int)`

```
static void insDirecta(int[] v) {  
    for (int i=1; i<=v.length-1; i++) {  
        int x=v[i]; //Elemento a insertar  
        int j=i-1; //Final parte ordenada  
        while (j >= 0 && v[j] > x) {  
            v[j+1]=v[j]; //copiar a la derecha  
            j--; //Seguir buscando a la izq.  
        } //end while  
        v[j+1] = x; //Insertar elemento  
    } //end for  
} //end insDirecta
```

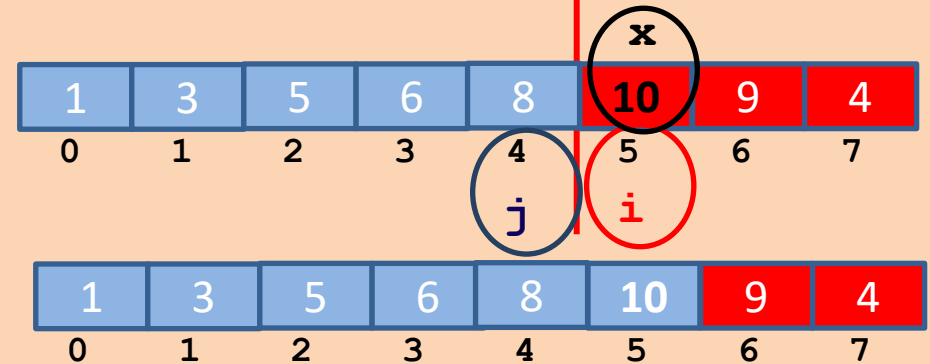
etapa i=4



# Algoritmo de inserción directa: insDirecta

etapa i=5

**v**



```
static void insDirecta(int[] v) {  
    for (int i=1; i<=v.length-1; i++) {  
        int x=v[i]; //Elemento a insertar  
        int j=i-1; //Final parte ordenada  
        while (j >= 0 && v[j] > x) {  
            v[j+1]=v[j]; //copiar a la derecha  
            j--; //Seguir buscando a la izq.  
        } //end while  
        v[j+1] = x; //Insertar elemento  
    } //end for  
} //end insDirecta
```

# Algoritmo de inserción directa: `orderedList`

```
import java.util.*;
```

```
public class FiguresGroup {  
    private static final int NUM_FIGURES = 10;  
    private Figure[] figuresList = new Figure[NUM_FIGURES];  
    private int numF = 0;
```

```
    public List<Figure> orderedList () {  
        List<Figure> l = ...  
        ...  
    }
```

## Ejercicio 3

### Figure

```
int compareTo(Figure f)  
...
```

```
l.get(j) > x
```

### List<E>

```
void add(int i, E e)  
void add(E e)  
E get(int i)  
int size()  
...
```

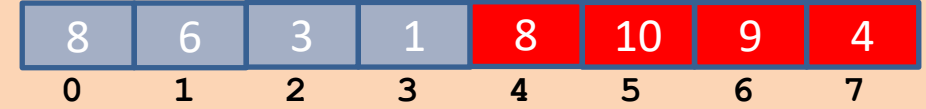
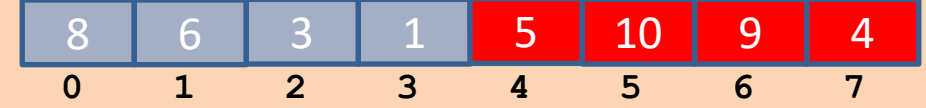
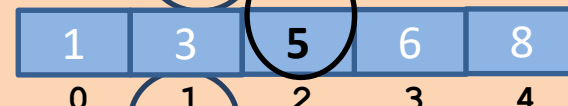
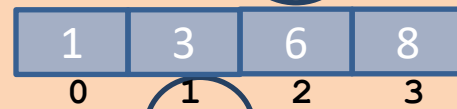
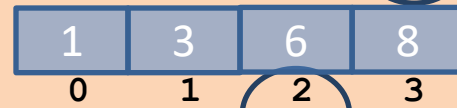
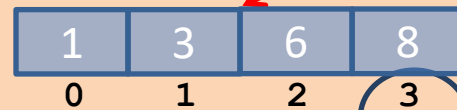
etapa  $i=4$

parte ordenada

parte a ordenar

1

figuresList





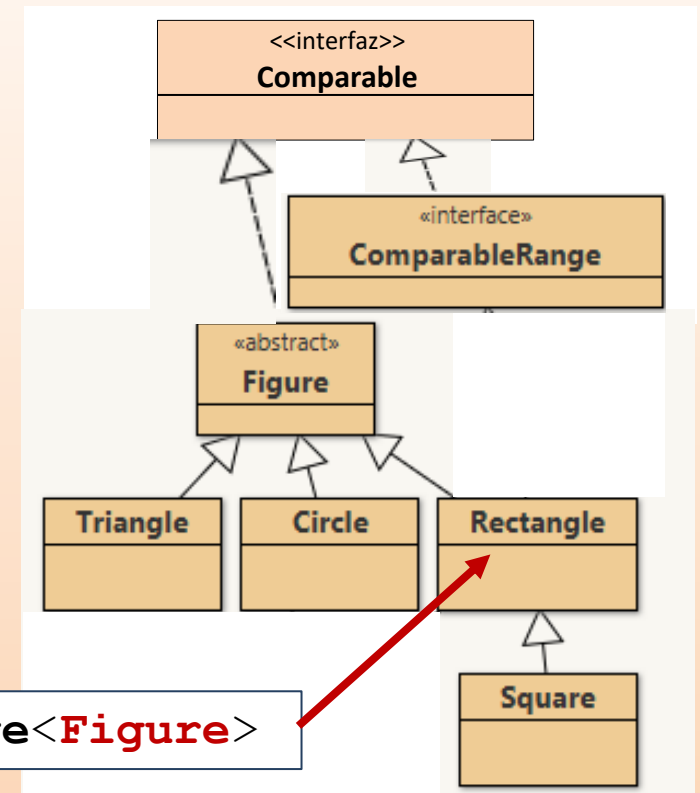
### 3. Extensión de una interfaz: ComparableRange

Las interfaces solo pueden heredar de otras interfaces. Las derivadas de una interfaz añaden funcionalidad a las especificaciones que heredan.

**Ejercicio 4:** Escribe una nueva interfaz **ComparableRange** que extienda la interfaz **Comparable** con el método:

```
int compareToRange (T o)
```

```
...implements ComparableRange<Figure>
```



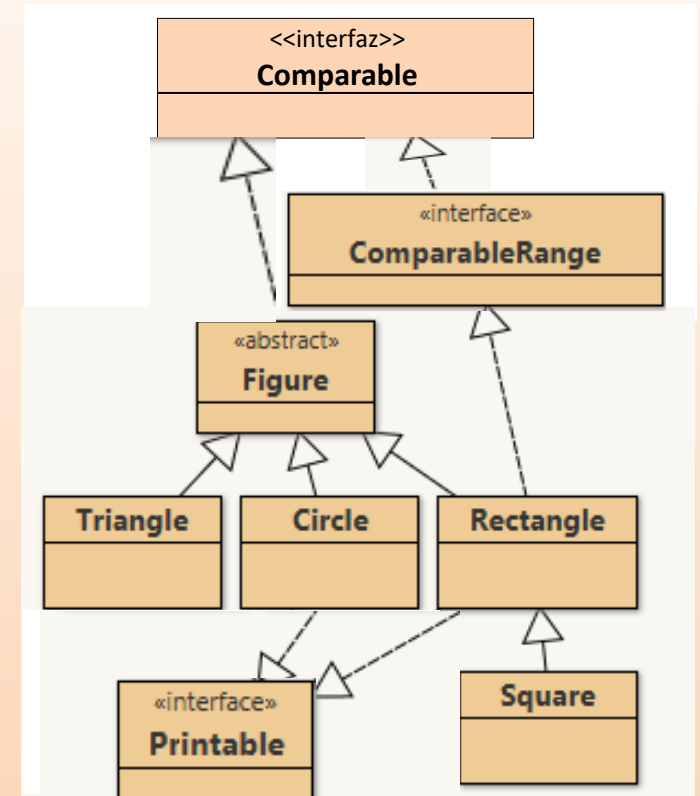
**Ejercicio 5:** Haz que la clase **Rectangle** implemente la interfaz **ComparableRange** teniendo en cuenta que solo se quiere usar esta comparacion para comparar pares de rectángulos y/o cuadrados. La norma para esta clase es que se comporte de forma similar a su clase padre excepto que considera iguales dos figuras si la diferencia de sus áreas en valor absoluto es menor o igual al 10% de la suma de sus áreas. Si son diferentes bajo este criterio, se comparan igual que el método **compareTo** que hereda.

### 3. Diseño de una interfaz: Printable

**Ejercicio 6:** Define una interfaz de nombre **Printable** que especifique un método de perfil:

```
void print(char c)
```

**Ejercicio 7:** Implementa la interfaz **Printable** en las clases **Circle** y **Rectangle**. Aprecia que también será posible dibujar cuadrados, dado que **Square** heredara de **Rectangle** el método **print** (código disponible en “fragmentos.java”).



#### Código para dibujar círculos

```
int n = (int) radius;
for (int j = 0; j <= n * 2; j++) {
    for (int i = 0; i <= n * 2; i++) {
        if (Math.pow(i-n, 2.0) + Math.pow(j-n, 2.0)
            <= (int) Math.pow(n, 2)) {
            System.out.print(c);
        }
        else { System.out.print(" "); }
    }
    System.out.println();
}
```

#### Código para dibujar rectángulos

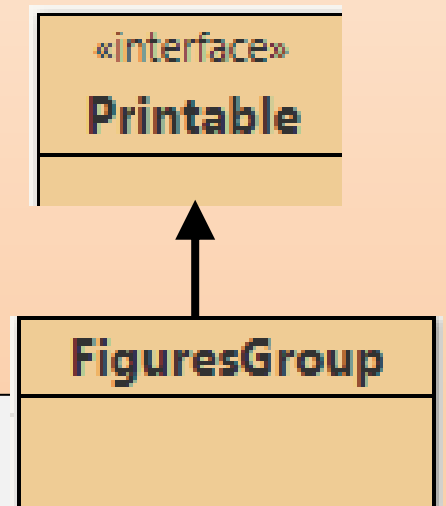
```
int b = (int) base;
int h = (int) height;
for (int i = 0; i < h; i++) {
    for (int j = 0; j < b; j++) {
        System.out.print(c);
    }
    System.out.println();
}
```

# Imprimir un grupo de figuras

**Ejercicio 8:** Implementa la interfaz **Printable** en la clase **FiguresGroup**. Debes partir de esta anterior implementación del método **print**, corrigiendo el código para que compile y funcione correctamente. Ten en cuenta que los elementos de **figuresList** son de tipo **Figure** y que no todos se pueden dibujar. Así que, además de comprobar que la clase de una determinada figura del grupo implementa **Printable**, debes facilitar el acceso al método **print**, pues no está implementado en la clase **Figure** que es el tipo de **figuresList**.

```
public void print(char c) {  
    for (int i = 0; i < numF; i++) {  
        figuresList[i].print(c);  
    }  
}
```

Se produce un error de compilación, porque no todas las figuras implementan **Printable**



Para comprobar que funciona, puedes aplicar el método **print** al grupo definido en la clase programa **FiguresGroupUse**.