

Tema 5 Diseño y definición de clases

- Definición.
- Declaración. Ejemplos.
- Tipos de métodos.
- Sobrecarga de métodos.
- El uso de this.
- Atributos static.
- Métodos static.
- Diseño de clases.

Definición de clase

- Es el **elemento básico** de la P.O.O.
- Una **clase** es una descripción de la estructura y el comportamiento de los objetos que pertenecen a ella.

```
[ cualificadores ] class NombreClase [ extend nombre_clase1 ] {
```

```
[ cualificadores ] tipo nomVar1;
```

```
[ cualificadores ] tipo nomVar2;
```

```
.....  
[ cualificadores ] tipo nombreMétodo1 ( [ lista_de_argumentos ] ) {  
    cuerpo  
}
```

```
[ cualificadores ] tipo nombreMétodo2 ( [ lista_de_argumentos ] ) {  
    cuerpo  
}
```

```
[ cualificadores ] tipo nombreMétodo13( [ lista_de_argumentos ] ) {  
    cuerpo  
}  
}
```

Declaración de una clase

```
class NombreClase {  
    atributos  
    métodos  
}
```

Una declaración de este tipo indica que la clase no descende de ninguna otra, aunque en realidad todas las clases declaradas en un programa Java, descenden directa o indirectamente de la clase **Object** que es la raíz de toda la jerarquía de clases en Java.

El cuerpo de la clase

Una vez declarada la clase, se declaran los **atributos y los métodos** de la misma.

Declaración de clase {

 Declaración de atributos

 Declaración de métodos

}

Atributos de objeto.- Son variables u objetos que almacenan valores **distintos** para instancias distintas de la clase (para objetos distintos).

Atributos de clase.- Son variables u objetos que almacenan el **mismo** valor para todos los objetos instanciados a partir de esta clase. (**static**)

Declaración de atributos

- ♦ Si no se especifica lo contrario los atributos **son de objeto** y no de clase.

Declaración :

tipo nombreAtributo

- ♦ Para declarar un **atributo de clase** se utiliza la palabra reservada **static**.

```
class Punto {  
    double x;    // abscisa del punto  
    double y ;  // ordenada del punto  
}
```

- x e y son variables asociadas a cada **objeto** de la clase Punto que se pueda **crear**.
- Se denominan **atributos** o **variables de instancia**.

Ejemplo. Definición básica

```
class PruebaPunto {  
    public static void main (String arg[ ] ) {  
        double  distanOrigen ;  
        Punto p1 = new Punto ( ) ;  
        Punto p2 = new Punto ( ) ;  
        Punto vp[ ] = new Punto [ 100 ] ;  
        p1.x = -1.0;    p1.y = -1.0 ;  
        p2.x = 1.0;    p2.y = 1.0 ;  
        distanOrigen = Math.sqrt(p2.x*p2.x+p2.y*p2.y );  
        System.out.println("Distancia al origen "+distanOrigen) ;  
    }  
}
```

Nueva definición de la clase Punto

```
class Punto {  
    private double x;           // abscisa del punto  
    private double y;          // ordenada del punto  
    public Punto () { }  
  
    public void asignar (double abs, double ord){  
        x= abs;  
        y= ord;  
    }  
    public double distanOrigen() {  
        return (Math.sqrt (x*x + y*y ) );  
    }  
    public double abscisa () {  
        return x; }  
    public double ordenada () {  
        return y; }  
} // fin de la clase
```

- Toda la información declarada **private** es exclusiva del objeto e inaccesible desde fuera de la clase.
- Toda la información declarada **public** es accesible desde fuera de la clase. Por defecto variables y métodos son **public**.
- En cualquier clase existe por defecto un método **sin tipo** cuyo nombre es el de la propia clase.
- Este es el método **constructor** que se utiliza para crear un nuevo objeto con el operador **new**.

Tipos de métodos

Los métodos se pueden clasificar según la función que realizan respecto al objeto en:

- **Constructores.** Permiten construir el objeto. Punto()
- **Modificadores.** Permiten alterar el estado del objeto.
- **Consultores.** Permiten conocer, sin alterar el estado del objeto.

Tipos de métodos ejemplo.

```
class Punto {  
    private double x; // abscisa del punto  
    private double y; // ordenada del punto
```

Constructor

```
    public Punto () {}
```

```
    public void asignar (double abs, double ord){
```

```
        x= abs;
```

```
        y= ord;
```

```
    }
```

```
    public double distanOrigen() {
```

```
        return (Math.sqrt (x*x + y*y ) ;
```

```
    }
```

```
    public double abscisa () {
```

```
        return x;
```

```
    }
```

```
    public double ordenada (){
```

```
        return y;
```

```
    }
```

```
} // fin de la clase
```

Modificador cambia los valores de las variables de instancia.

Es posible definir un método constructor, que además de crear el objeto, altere su estado, por ejemplo para inicializarlo

Utilización de la clase punto.

Comentarios.

Para poder utilizar un método correspondiente a un objeto, utilizaremos la notación de punto.

```
class Prueba {  
    public static void main ( String arg [ ] ) {  
        Punto p = new Punto();  
        double dist ;  
        p.y=5 ;    // sería un error, por ser información privada.  
        p.asignar( 1, 5);  
        dist =p.distOrigen();  
        System.out.println( " Distancia : "+dist ) ;  
    }  
}
```

Sobrecarga de métodos

- Se denomina sobrecarga, a la definición de un mismo elemento (símbolos, identificador..) con distintos significados, y en función de cómo se utilice puede interpretarse su significado.
- Ejemplo. **El operador +**

```
a+=5;  
System.out.print("vale :"+ a);
```

Sobrecarga de métodos.

```
class Punto {  
    private double x;          // abscisa del punto  
    private double y;          // ordenada del punto  
    public Punto () { }        // primer constructor  
    public Punto ( double abs , double ord ) { //segundo constructor  
        x= abs;  
        y= ord ; }  
    public Punto ( double coord ) {          // tercer constructor  
        x= coord;  
        y= coord ; }  
  
    public void asignar (double abs, double ord){  
        x= abs; y= ord; }  
    .....  
} // fin de la clase
```

- En este ejemplo tenemos tres métodos constructores (**Punto**) que sólo se diferencian por su argumento.

Sobrecarga de métodos

- El lenguaje seleccionará uno u otro, dependiendo del argumento que utilicemos en la llamada al método.

Ejemplo.

```
class Prueba {  
    Punto p= new Punto( );  
    Punto p1 = new Punto (1.0, -1.0);  
    Punto p2 = new Punto (1.0) ;  
  
    p.asignar(1.0 , -1.0) ;  
    .....  
}
```

El uso de this.

- **Java** incluye una referencia especial denominada **this** que se utiliza dentro de cualquier método **para hacer referencia al objeto actual**.

Ejemplo.

```
class Punto {  
    private double x ;  
    private double y ;  
    .....
```

```
// usamos el segundo constructor, pero a sus parámetros también les  
// llamamos x e y
```

```
public Punto (double x, double y) {
```

```
    this.x = x;
```

```
    this.y = y ;
```

```
}
```

```
}
```

← Parámetro formal

↗ Atributo de clase

Modificadores de clases

modificador **class** NombreClase [**extends** NombreSuperclase]
[**implements** listaDeInterfaces]

- Son **palabras reservadas** que se anteponen a la declaración de la clase.
 - Si no se especifica ningún modificador , la clase será visible en todas las declaradas en el mismo paquete.
 - Si no se especifica ningún paquete, se considera que la clase pertenece a un paquete por defecto al que pertenecen todas las clases que no declaran explícitamente el paquete al que pertenecen.
- Los modificadores posibles son:
 - **public**
 - **abstract**
 - **final**

public

- Cuando se crean varias clases que se agrupan formando un paquete (package), **sólo** las declaradas **public** pueden ser **accedidas** desde otro **paquete**.
- Toda clase **public** debe ser declarada en un **fichero fuente** con el **nombre** de esa clase pública:
NombreClase.java
- *En un **fichero fuente** puede haber más de una clase, pero **sólo una** con el modificador **public**.*

abstract

- Las clases abstract no pueden ser instanciadas.
- Sirven para **declarar subclases** que deben redefinir los métodos declarados abstract.
- Los métodos de una clase abstract pueden no ser abstract.
- En este último caso, tampoco se podrán declarar objetos de una clase declarada como abstract.
- Cuando existe algún **método abstract**, la clase debe ser **declarada abstract** , en caso contrario el compilador dará un error.

Ejemplo: abstract

```
abstract class Animal {
```

```
    String nombre;
```

```
    int patas;
```

```
    public Animal(){}
```

```
    public Animal (String n, int p ) {
```

```
        nombre= n;
```

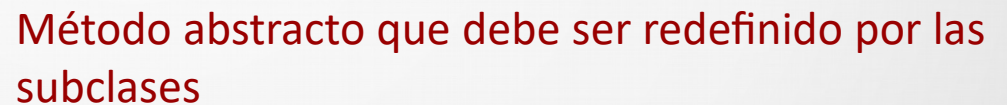
```
        patas = p;
```

```
    }
```

Constructor



Método abstracto que debe ser redefinido por las subclases

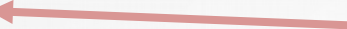


```
abstract void habla ();
```

```
}
```

Ejemplo: abstract

```
class Perro extends Animal {  
  // La clase Perro es una subclase de la clase abstract Animal  
  String raza ;  
  public Perro (String n, int p, String r ) {  
    Super (n, p );  
    raza = r; }  
  public void habla () {  
    System.out.println ("Me llamo " +nombre+ " :GUAU " );  
    System.out.println ("Mi raza es " +raza ) ;  
  } // cierra el método  
} // cierra la clase
```



Llama al constructor de la Superclase

Este método es necesario redefinirlo para poder instanciar objetos de la clase Perro.

Ejemplo: abstract

```
class EjemAbstracta {  
    public static void main ( String argm[ ] ) {  
        Perro toby ; // declaramos el objeto toby de la clase Perro  
        toby = new Perro ( "Toby", 4 , "San Bernardo");  
        // creamos el objeto  
        /* instanciamos el objeto toby, para que reciba  
        los parámetros y métodos de la clase Perro. */  
    }  
}
```


final

- Una clase declarada ***final*** impide que pueda ser superclase de otras clases. Ninguna clase puede heredar de una clase ***final***.
- A diferencia del abstract, pueden existir en la clase **métodos *final***, sin que la clase que los contiene sea final.
- Una clase puede ser a la vez:
 - ❖ **public abstract**
 - ❖ **public final**

En resumen

- **Declaración de una clase**

`class NombreClase`

- **Modificadores de clase**

public : Fichero fuente con el nombre de la clase.

abstract: No se podrán crear objetos de esta clase.

final: No puede ser superclase de otras.

Es decir , no se puede heredar.

Prácticas

- Ejercicios:

Realiza un programa que solicite al usuario el nombre y edad de un alumno, y después cree un objeto de la clase Alumno invocando al constructor de dicha clase.

En la última sentencia muestra los datos almacenados en el objeto alumno.

Prácticas

- Ejercicios:

Realiza un programa que solicite al usuario el nombre, telefono y edad de una persona, y después cree un objeto de la clase Persona , que tiene estos tres atributos con el modificador private, pasando al constructor los valores como constantes.

Además de los correspondientes métodos get y set , la clase Persona tendrá un **método** que devuelva una cadena con todos los atributos del objeto.

Declaración de atributos

- ❖ Los atributos sirven para almacenar valores de los objetos que se instancian a partir de una clase.

- Sintaxis

[modifiDeAmbito] [static] [final][transient] [volatile] tipo nombreAtributo

Tipos de atributos

- ✓ static
- ✓ final
- ✓ transient
- ✓ volatile

static

- Crea un atributo/variable de clase.
- Significa que todas las instancias de la clase (objetos instanciados de la clase), contienen la misma variable y cuyo valor es el mismo para todos los objetos.
- **Si un objeto modifica el valor, quedará modificado para todos los objetos**
- Se accede a ellos mediante **nombreClase.nomAtributo**

final

- Indica que el **atributo se comporta como una constante**.
- Si se intenta modificar el valor del atributo, desde el código de la aplicación, se genera error de compilación.
- Puede ir unido a `static`, creando una constante de clase.

transient

- Marca al atributo como transitorio, para no ser serializado. Lo emplearemos en java beans.

Volatile y synchronized

- Se utilizan para mecanismos de sincronización en java.
- volatile es más simple y más sencillo que synchronized, lo que implica también un mejor rendimiento. Sin embargo volatile, a diferencia de synchronized, no proporciona atomicidad, lo que puede hacer que sea más complicado de utilizar. Es un atributo accedido de forma asíncrona mediante hilos.
- Para indicar al compilador que es posible que este atributo vaya a ser modificado por varios threads de forma simultánea y asíncrona, y que no queremos guardar una copia local del valor para cada thread a modo de caché, sino que queremos que los valores de todos los threads estén sincronizados en todo momento, asegurando así la visibilidad del valor actualizado a costa de un pequeño impacto en el rendimiento.

Atributos static - Ejemplo -

static tipo nombreAtributo

```
class Persona {  
    static int numPersonas = 0 ;           // atributo de clase  
    String nombre;                         // atributo de objeto  
    public Persona (String n) {           // método constructor  
        nombre =n;  
        numPersonas++; }  
    public void muestra () {  
        System.out.print ("Soy "+nombre);  
        System.out.print("pero  hay "+(numPersonas-1) + " personas  
        más");  
    }  
}
```

Ejercicio:

- Incluir el atributo statico numAlum en vuestra clase Persona

modificadores de ámbito de atributos y métodos

- **private**
- **public**
- **protected**
- **El ámbito por defecto (friendly)**

private

- Solo pueden ser accedidos desde dentro de la clase.
- No son accesibles desde las subclases de esa clase.
- Permite la encapsulación u ocultación de datos.
- Se accederá a los atributos mediante métodos consultores o modificadores. (**get y set**)

public

- Podrán ser referenciados por cualquier clase, incluso las de fuera del paquete.
- Usar cuando deban ser visibles para todo el mundo.

protected

- Será accesible a la clase y a sus clases derivadas (o subclases) si están en el mismo paquete.
- En general, si una subclase no se encuentra en el mismo paquete que la superclase, no tendrá acceso a atributos/métodos protegidos.
- **Usar si se quiere que las clases del mismo paquete puedan acceder a atributos/métodos**

friendly --package

- Serán referenciados solamente por las clases del mismo paquete.
- Usar cuando los atributos/métodos deban estar ocultos fuera del paquete.
- Usar `package` para agrupar las clases en paquetes.
- La diferencia con *protected* es que ésta es heredada por las subclases de paquetes diferentes.

Modificadores de visibilidad aplicados a clase, atributos v métodos

Visibilidad	public	protected	default	private
Desde la Misma Clase,	SÍ	SÍ	SÍ	SÍ
Desde cualquier clase en el mismo paquete.	SÍ	SÍ	SÍ	NO
Desde cualquier clase fuera del paquete.	SÍ	NO	NO	NO
Desde una subclase en el mismo paquete.	SÍ	SÍ	SÍ	NO
Desde una subclase fuera del mismo paquete.	SÍ	SÍ	NO	NO

A light blue rounded square with the word "public" in a white, sans-serif font.

public

Los Métodos declarados **public** en una superclase deben ser también **public** en todas las subclases.

A teal rounded square with the word "protected" in a white, sans-serif font.

protected

Los Métodos declarados **protected** en una superclase, deben bien ser **public** o **protected** en sus subclases. No pueden ser **private**.

A purple rounded square with the word "default" in a white, sans-serif font.

default

Los Métodos declarados sin control de acceso (**default**) no pueden ser declarados como **private**.

Un método sobrescrito no puede ser más restrictivo que el original. Si más público.

Métodos static

- Los métodos static son métodos de clase (no de objeto) y por tanto, no necesitan instanciar la clase (crear un objeto de esa clase) para poder llamar a ese método.
- Se han estado utilizando hasta ahora siempre que se declaraba una clase ejecutable, ya que para poder ejecutar el método `main()` no se declara ningún objeto de esa clase.
- Los métodos de clase **(static)** únicamente pueden acceder a sus atributos de clase **(static)** y nunca a los atributos de objeto **(no static)**.

Ejemplo:

```
class EnteroX {  
    int x;
```

```
    static int getX() {  
        return x;  
    }
```

```
    static void setX(int nuevaX)  
    {  
        x = nuevaX;  
    }  
  
}
```

Mostraría el siguiente mensaje de error por parte del compilador:

MetodoStatic1.java:4: Can't make a static reference to nonstatic variable x in class EnteroX.

return x;

MetodoStatic1.java:7: Can't make a static reference to non static variable x in class EnteroX.

x = nuevaX;

2 errors.

Ejemplo:

Sí que sería correcto:

```
class EnteroX {  
    static int x;  
  
    static int getX() {  
        return x;  
    }  
    static void setX(int nuevaX) {  
        x = nuevaX;  
    }  
}
```

Al ser los métodos static, puede accederse a ellos sin tener que crear un objeto.

Entero X:

```
class AccedeMetodoStatic {  
    public static void main(String argm[ ]) {  
        EnteroX.setX(4);  
        System.out.println(EnteroX.x()); }  
}
```


Resumen

clases

Atributos

static

final

transient

volátil

Métodos

static

abstract

final

native

synchronized

public

abstract

final

Modificadores de ámbito

private

public

protected

amb. por defecto

Modificadores de ámbito

private

public

protected

amb. por defecto

Trabajando con objetos

- **Objetos y referencias.** La diferencia principal entre **valores primitivos y valores referencia** consiste, en que las variables que representan a los primeros, mantienen el valor de los mismos mientras que las segundas mantienen una referencia (dirección) a su posición real en memoria.
- **El operador “.”** El operador **punto** se emplea para seleccionar el método que se desee utilizar sobre el objeto en curso.

Trabajando con objetos

Ejemplo: Un Panel es un objeto gráfico en el que se sitúan elementos tales como botones, menús, etc

```
Panel p = new Panel() ;           //Creación de un "Panel" p
Button boton ;                    // boton vale null
boton = new Button() ;           // boton referencia al objeto creado
boton.setLabel("Ejemplo");        // Se pone etiqueta al botón
p.add(boton);                     //Se añade el botón al Panel
```

Asignación

La operación `v1 = v2;`

Asigna a `v1` del valor de `v2` reemplazando el contenido de `v1` con el de `v2`, tanto si ambas pertenecen a uno de los tipos primitivos como si se trata de variables referencia.

Con referencias, la asignación significa tan sólo un reemplazamiento de las referencias correspondientes, no del contenido referenciado. Ejemplo:

`Objeto oB1 = new Objeto();` *//oB1 referencia a un Objeto*

`Objeto oB2 = oB1;` *//oB1 y oB2 referencian al Objeto primero*

Tras la ejecución, se tiene un mismo objeto al que se puede nombrar de dos formas distintas: `oB1` y `oB2`.

Asignación

En el ejemplo, perdemos la referencia al segundo objeto:

```
Objeto oB1 = new Objeto();    //oB1 referencia a un Objeto  
Objeto oB2 = new Objeto();    //oB2 referencia a otro Objeto  
oB2 = oB1;                   //oB1 y oB2 referencian al Objeto primero
```

Igual que antes, oB1 y oB2 nombran al mismo objeto.

Pero ahora nada referencia al objeto creado en segundo lugar.
Cuando ninguna variable referencia a un objeto, se dice que dicho objeto está **desreferenciado**.

Asignación

Ejemplo . Se trata de crear dos botones en cierto panel ya definido p:

```
Button bot1 = new Button("BOT1");
```

```
Button bot2 = bot1;
```

```
bot2.setLabel("BOT2");
```

```
p.add(bot1);
```

```
p.add(bot2);
```

Asignación

- Tan sólo se ha creado un botón, por lo tanto, tras la asignación efectuada en la segunda línea se tienen dos variables que referencian un mismo objeto, el creado en la línea primera.
- Tanto bot1 como bot2 representan un único objeto. La operación `setLabel()` se efectúa en cada caso sobre un único objeto, por lo que cuando ambos botones se añadan en la componente gráfica (el Panel) aparecerán con el mismo rótulo (BOT2).
- Es posible operar sobre un objeto utilizando cualquiera de los nombres de variables que lo representen.
- Se tiene que ser cuidadoso cuando se dan situaciones de referenciación múltiple.

Copia

Copia de objetos. Puede parecer que, ya que la asignación entre objetos sólo supone una copia de las referencias, es imposible efectuar una copia de los objetos como tales.

- En primer lugar, si la estructura del objeto es conocida y accesible, es posible realizar dicha copia mediante una copia individual, elemento a elemento, de cada uno de los elementos del tipo primitivo correspondiente.
- Así, por ejemplo: Copia de un vector `vec1` de `N` valores de tipo `double`. Se debe crear un nuevo vector:

```
double vec2[] = new double[N];  
    for (int i=0; i<N;i++)  
        vec2[i] = vec1[i];
```


Trabajando con objetos objetos -- Copia

- En segundo lugar, existe un **método**, denominado **clone()**, definido para cualquier objeto que permite efectuar dicha copia. Este método hace un uso explícito de new. Así, para el caso anterior:
- copia de un vector vec1 de N valores de tipo double.
- **double vec2[] = (double []) vec1.clone();**
- El uso del método **clone** ha exigido una operación explícita de transformación de tipos (**casting**), esto es así porque clone sólo devuelve una referencia a un elemento sin estructura o tipo conocido. El casting se la otorga.

Trabajando con objetos

- **Paso de parámetros.**

Como sabemos, en Java el paso de parámetros es siempre por valor.

Cuando el parámetro de cierto método es un objeto, representado mediante una referencia, el valor que se transmite al método es la referencia a la zona de memoria donde se encuentra el objeto, no el del objeto en sí.

Naturalmente, cualquier modificación que se realice a través de la referencia al objeto original, implica una modificación del valor del objeto .

Trabajando con objetos: El operador ==

- El operador == .

En los **tipos primitivos** el operador == es cierto o falso según sean o no iguales los valores de las variables que se comparan.

Cuando este operador se aplica a variables que referencian a objetos devolverá cierto o falso según sean o no iguales **las referencias a dichos objetos**.

Por lo tanto, si se aplica el operador a dos objetos distintos, pero que contienen la misma información, el resultado que se obtendrá será false.

En Java, los contenidos de objetos distintos pueden compararse entre sí mediante el método **equals ()** para el que ya se ha visto algún uso con valores de tipo String.

Trabajando con objetos

Ocasionalmente, como en el caso de los Arrays pueden existir métodos específicos, como muestra el siguiente ejemplo:

```
.....
double v2[]
double v1[] = new double[100];
for (int i=0; i<100; i++)
v1[i]=i;
v2 = (double [])v1.clone();
double v3[] = v2;
if (Arrays.equals(v1,v1))
System.out.println("v1 igual v1");
if (Arrays.equals(v1,v2))
System.out.println("v1 igual v2");
if (Arrays.equals(v2,v3))
System.out.println("v1 igual v3");

if (v1== v1){
    System.out.println("v1==v1");    }
if (v1==v2){
    System.out.println("v1==v2");
} else System.out.println((v1==v2));

if (v2==v3) {
    System.out.println("v1==v3");    }
}
}
v1 igual v1
v1 igual v2
v1 igual v3
-----
v1==v1
false
v1==v3
```

Trabajando con objetos

Naturalmente, si se efectúa en las condiciones anteriores la comparación:

```
if (v1 == v2) { .... }
```

se evaluará a false, ya que v1 y v2 contienen referencias a distintas zonas de memoria, puesto que el método clone() hace un new para v2.

El garbague collector.

- Cuando para un objeto , creado en algún momento de la ejecución de un programa no tiene ninguna variable que lo referencie, decimos que dicho objeto está **desreferenciado** , es decir , que no es posible volver a operar con él.
- **Ejemplo.**
- Objeto oB1 = new Objeto(); //oB1 referencia a un Objeto
- Objeto oB2 = new Objeto(); //oB2 referencia a otro Objeto
- oB2 = oB1; //oB1 y oB2 referencian al Objeto primero

//nada referencia al objeto OB2

El garbague collector.

```
public class Vector {  
    private int i;  
    int v[ ];  
public Vector(int x) {  
    i=0;  
    v=new int [x]; }  
    void ponACero() {  
        for( i=0; i< v.length; i++) v[i]=0; }  
public void Ordena(int b){  
    boolean sw=false;  
        while (!sw){  
            sw=true;  
            for(int c=1;c<v.length-b;c++)  
                if(v[c-1]>v[c]) { int aux=v[c];  
                                v[c]=v[c-1];  
                                v[c-1]=aux;  
                                sw=false; }  
        }  
    }  
}
```

El garbague collector.

```
int QuitaRepes(Vector p){
    int l = p.v.length;
    int i,j,conta=0;
    for(int c=0; c<l;c++){
        if (p.v[c]>1) { i=0;
            while (v[i] != c) i++;
            for ( j=i+1;
                j<v.length;j++)
                v[j-1]=v[j];
            conta++;
            p.v[c]--;
            v[j-1]=0;
            c--; }
    }
    return conta; }
}
```