

- **Ejemplo:** la clase *Complejo* que se muestra a continuación, dispone del método de clase *anularModuloMenor* cuyos parámetros son dos objetos de la clase *Complejo* y cuya función es anular (poner a cero) el complejo que tenga el módulo menor.

```
public class Complejo
```

```
{
```

```
    private double re , im; —————> Atributos privados que almacenan la parte real (r) y  
                                         la parte imaginaria (i) del número complejo
```

```
    public Complejo(double r, double i) —————> Constructor con parámetros
```

```
{
```

```
    re = r ;
```

```
    im = i ;
```

```
}
```

```
    public void setRe(double r) { re = r; } —————> Método para establecer la parte real
```

```
    public void setIm(double i) { im = i; } —————> Método para establecer la parte imaginaria
```

```
    public double getRe() { return re; } —————> Método para consultar la parte real
```

```
    public double getIm() { return im; } —————> Método para consultar la parte imaginaria
```

# Métodos (XII)



## CLASES

```
public double modulo() { return Math.sqrt (re * re + im * im) ; }
```

→ Método para  
calcular el módulo

```
public void anular() → Método para anular el complejo  
{ (poner a cero sus componentes)
```

```
    re = 0.0;
```

```
    im = 0.0;
```

```
}
```

```
public static void anularMenorModulo(Complejo c1, Complejo c2) → Método de clase para anular el  
{ complejo que tenga el módulo menor
```

```
    double moduloc1, moduloc2; → Variables locales para almacenar los módulos de los dos complejos
```

```
    moduloc1 = c1.modulo();
```

```
    moduloc2 = c2.modulo();
```

} → Cálculo de los módulos

```
    if (moduloc1 < moduloc2 )
```

```
        c1.anular();
```

```
    else c2.anular();
```

} → Anulación del complejo de menor módulo

```
}
```

```
public String toString() → Método que devuelve el valor del complejo en formato String
```

```
{
```

```
    String param, ReString, ImString;
```

```
    ReString = Double.toString(re); → Representación en String de la componente real
```

```
    ImString = Double.toString(im); → Representación en String de la componente imaginaria
```

```
    param = ReString + " + " + ImString + " i ";
```

```
    return param;
```

```
}}
```

# Métodos (XIII)



## CLASES

Se construyen dos objetos de la clase *Complejo* y se muestran sus valores por pantalla. Posteriormente se llama al método *anularModuloMenor* y se vuelven a mostrar los objetos por pantalla para comprobar que los cambios hechos desde el método quedan reflejados en el objeto.

```
public class MainClass
{
```

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

```
        Complejo complejo1 = new Complejo(2.0 , 2.0);
```

```
        Complejo complejo2 = new Complejo (1.0 , 1.0);
```

```
        ❶ System.out.println("c1 = " + complejo1.toString());
```

```
        ❷ System.out.println("c2 = " + complejo2.toString());
```

```
        Complejo.anularMenorModulo(complejo1, complejo2);
```

```
        ❸ System.out.println("c1 = " + complejo1.toString());
```

```
        ❹ System.out.println("c2 = " + complejo2.toString());
```

```
    }
```

```
}
```

❶  
❷  
❸  
❹

Creación de los dos objetos de la clase *Complejo*

Mostrar por pantalla los valores de los dos complejos

Anular el complejo con menor módulo

Mostrar por pantalla los valores de los dos complejos

- **Ejemplo:** la clase *MainClass* implementa el método de clase *escalarMatriz1D* que tiene un parámetro de tipo matriz y un segundo parámetro de tipo **float**. La función del método es multiplicar todos los elementos de la matriz por el segundo parámetro.

```
public class MainClass
{
    public static void escalarMatriz1D (float[] m1D, float factor) → Método para multiplicar los elementos
    {
        int tam = m1D.length; → Obtener el tamaño de la matriz
        for (int i = 0 ; i<tam ; i++) } → Recorrer la matriz y multiplicar cada
        m1D [i] *= factor;           elemento por el factor de escalado
    }
    public static void mostrarMatriz1D (float[] m1D) → Método para mostrar por pantalla el
    {
        int tam = m1D.length; → Obtener el tamaño de la matriz
        System.out.print("[ ");
        for (int i = 0 ; i<tam ; i++) } → Recorrer la matriz y mostrar cada
            System.out.print(" " + m1D[i]); uno de los elementos por pantalla
        System.out.println(" ]");
    }
    // ...
}
```

El método **main** declara e inicializa una matriz 1D, y muestra por pantalla el contenido de la matriz. Después llama al método *escalarMatriz* y por último vuelve a mostrar los elementos de la misma. Se comprueba que las operaciones realizadas sobre la matriz dentro del método quedan reflejadas en ella, ya que las matrices son pasadas siempre por referencia.

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

```
{
```

```
    float[] matriz1D = {1,2,3,4}; —————→ Declaración e inicialización de la matriz 1D
```

```
    ❶ mostrarMatriz1D(matriz1D); —————→ Presentación en pantalla del contenido de la matriz
```

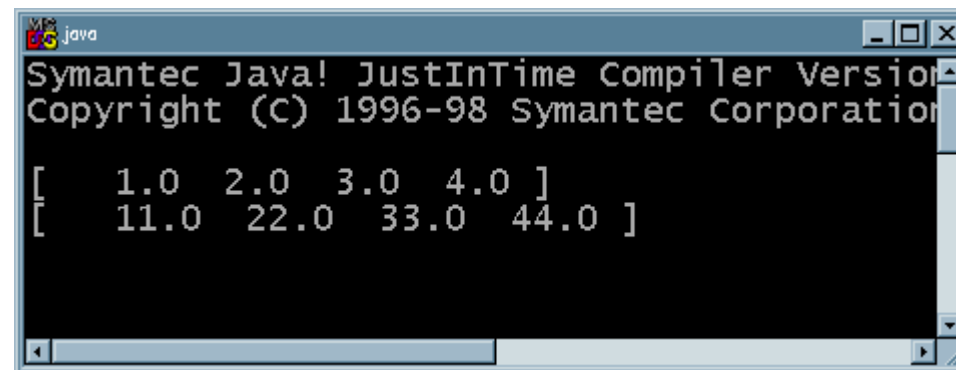
```
    escalarMatriz1D(matriz1D, 11); —————→ Multiplicación de los elementos de la matriz por 11
```

```
    ❷ mostrarMatriz1D(matriz1D); —————→ Presentación en pantalla del contenido de la matriz
```

```
}
```

```
}
```

❶  
❷



- **Ejemplo:** la clase *MainClass* implementa el método *duplicarPrimitivo* que tiene un parámetro primitivo de tipo **int**, y el método *duplicarPrimitivoEnMatriz* que tiene un parámetro de tipo **matriz int**. En el método **main** se comprueba que el primer método no puede modificar el valor de su parámetro por ser un tipo primitivo y pasarse por valor al método, y el segundo sí puede hacerlo por ser un tipo matriz y pasarse por referencia.

```
public class MainClass
{
    public static void duplicarPrimitivo (int i)
    {
        i *= 2 ;
    }
    public static void duplicarPrimitivoEnMatriz(int[] i)
    {
        i[0] *= 2 ;
    }
    // ...
}
```

# Métodos (XVII)



## CLASES

```
public class MainClass
{
    public static void main (String[] args)
    {
        int entero=10;
        ❶ System.out.println("Valor inicial del entero primitivo = " + entero);
        duplicarPrimitivo(entero);
        ❷ System.out.println("Valor final del entero primitivo = " + entero);
        int[] enteroMatriz = { 10};
        ❸ System.out.println("Valor inicial del entero primitivo en matriz = " + enteroMatriz[0]);
        duplicarPrimitivoEnMatriz(enteroMatriz);
        ❹ System.out.println("Valor final del entero primitivo en matriz = " + enteroMatriz[0]);
    }
}
```

```
java
Symantec Java! JustInTime Compiler Version 3.10.088(i) f
Copyright (C) 1996-98 Symantec Corporation

❶ Valor inicial del entero primitivo = 10
❷ Valor final del entero primitivo = 10
❸ Valor inicial del entero primitivo en matriz = 10
❹ Valor final del entero primitivo en matriz = 20
```

# Tabla resumen de los modificadores de acceso



## CLASES

- Los modificadores de acceso **public**, **protected**, y **private** aplicables a las clases, las variables miembro y los métodos se resumen en la siguiente tabla.  
La columna **default** describe el tipo de acceso permitido cuando no se especifica ningún modificador.

Componente	public	protected	private	default
Desde la propia clase	SÍ	SÍ	SÍ	SÍ
Desde otra clase del mismo paquete	SÍ	SÍ	NO	SÍ
Desde otra clase de otro paquete	SÍ	NO	NO	NO
Desde una subclase del mismo paquete	SÍ	SÍ	NO	SÍ
Desde una subclase de otro paquete	SÍ	SÍ	NO	NO



- Propiedades (i)
  - Método especial que se llama automáticamente siempre que se crea un objeto de una clase. Su principal función es iniciar los atributos.
  - Su nombre es el mismo que el de la clase a la que pertenece.
  - No tienen valor de retorno.
  - No se heredan.
  - Excepto en casos excepcionales los constructores se declaran públicos.
  - En el cuerpo del constructor se puede asignar valores a los atributos, invocar métodos de la clase, o llamar a métodos de otras clases.
  - Una clase puede definir varios constructores que se diferencian en el tipo y número de argumentos (caso de sobrecarga de métodos). De esta manera se puede iniciar un objeto de distintas formas. Se llama **constructor por defecto** al constructor que no tiene argumentos.
  - El compilador proporciona a todas las clases declaradas un constructor público por defecto (sin parámetros), y que no hace nada, pero que es necesario porque cada vez que se crea un nuevo objeto se llama al constructor de la clase.

- Propiedades (ii)
  - Si el constructor de una clase es **private**, sólo un método **static** de la propia clase puede crear objetos. (Ejemplo: patrón Singleton)
  - Si una clase define un constructor (con o sin parámetros), el constructor por defecto desaparece, y si se quiere utilizar un constructor sin parámetros hay que definirlo explícitamente.  
**Ejemplo:** en la figura se muestra parte de la clase *Cuadrado*, que define un constructor con un parámetro. Si se intenta crear un nuevo objeto de la clase *Cuadrado* utilizando la sentencia `Cuadrado elCuadrado = new Cuadrado()` el compilador muestra el mensaje de error *"No constructor matching Cuadrado() found in Cuadrado"* porque ya no existe un constructor por defecto sin parámetros ya que ha sido anulado con la definición del constructor de un parámetro.

```
public class Cuadrado
{
    private double lado;
    public Cuadrado(double l) { lado=l; }
}
```

→ Constructor con un parámetro ⇒ el constructor por defecto se anula.

- Propiedades (iii)
  - Un constructor puede llamar a otro constructor previamente definido en la misma clase utilizando la palabra reservada **this** y debe ser siempre la primera sentencia.  
**Ejemplo:** la clase *Circulo* define un constructor con parámetros, y el constructor por defecto (sin parámetros) realiza una llamada al primero.

```
public class Circulo {  
    private double cx, cy, radio;  
    public Circulo(double x, double y, double r)  
    {  
        cx = x;  
        cy = y;  
        radio = r;  
    }  
    public Circulo()  
    {  
        this (0.0, 0.0, 1.0);  
    }  
}
```

- Proceso de creación de objetos en Java
  - Para crear un objeto de una clase hay que utilizar el operador **new** y un constructor de la clase.

`nombreClase nombreObjeto = new nombreClase( [parámetros] )`

- La secuencia de acciones es la siguiente:
  - Java reserva automáticamente la memoria necesaria para ubicar el objeto. Si no hay espacio suficiente **new** lanza la excepción **OutOfMemoryError**.
  - Se inicializan los atributos del objeto, a sus valores iniciales (si han sido iniciados en su declaración) o a los valores por defecto del sistema: cero para las variables numéricas, nulo para el tipo **char**, **false** para **boolean** y **null** las referencias a objetos.
  - Se llama al constructor de la clase. Si la clase no define explícitamente un constructor, se llama al constructor por defecto que proporciona el compilador automáticamente a cada clase. Como una clase puede definir múltiples constructores, el constructor que se invoca en la creación del objeto es seleccionado con las reglas que se utilizan en la selección de métodos sobrecargados.

# Constructores (V)



## CLASES

- **Ejemplo:** creación de un objeto perteneciente a una clase que no proporciona constructor.

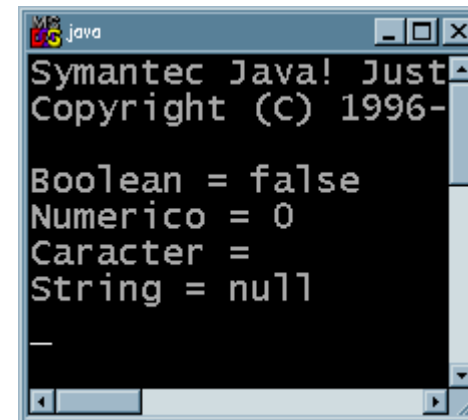
```
public class Ejemplos
{
    public static void main (String[] args)
    {
        Prueba pru = new Prueba();
        System.out.println("Boolean = " + pru.b);
        System.out.println("Numerico = " + pru.i);
        System.out.println("Caracter = " + pru.c);
        System.out.println("String = " + pru.s);
    }
}
```

1. Se reserva espacio para el objeto *pru*.
2. Se inicializan las variables miembro a los valores por defecto.
3. Se llama al constructor por defecto proporcionado por Java (que no hace nada).

Las variables miembro quedan inicializadas a los valores por defecto

```
public class Prueba
{
    public boolean b;
    public int i;
    public char c;
    public String s;
}
```

Clase sin constructor



# Constructores (VI)



## CLASES

- **Ejemplo:** creación de un objeto perteneciente a una clase que no proporciona constructor y en la que las variables miembro son iniciadas en su propia declaración.

```
public class Ejemplos
{
    public static void main (String[] args)
    {
        Prueba pru = new Prueba();
        System.out.println("Boolean = " + pru.b);
        System.out.println("Numerico = " + pru.i);
        System.out.println("Caracter = " + pru.c);
        System.out.println("String = " + pru.s);
    }
}
```

1. Se reserva espacio para el objeto *pru*.
2. Se inicializan las variables miembro a los valores de inicio especificados en la declaración de las variables.
3. Se llama al constructor por defecto proporcionado por Java (que no hace nada).

Las variables miembro quedan inicializadas a los valores de inicio

```
public class Prueba
{
    public boolean b = true;
    public int i = 8;
    public char c = 'a';
    public String s = "cadena";
}
```

Clase sin constructor

Valores de inicio de las variables miembro

```
Symantec Java! JustI
Copyright (C) 1996-9
Boolean = true
Numerico = 8
Caracter = a
String = cadena
```

# Constructores (VII)



## CLASES

- **Ejemplo:** creación de un objeto perteneciente a una clase que proporciona constructor.

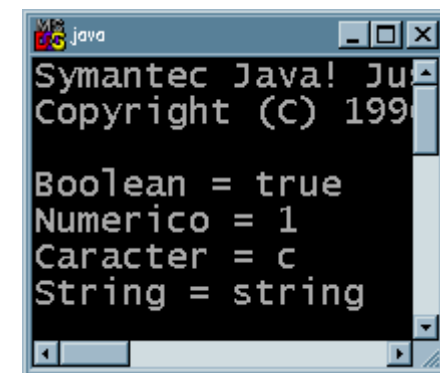
```
public class Prueba
{
    public boolean b;
    public int i;
    public char;
    public String s;

    public Prueba(boolean b, int i, char c, String s)
    {
        this.b = b;
        this.i = i;
        this.c = c;
        this.s = s;
    }
}
```

```
public class Ejemplos
{
    public static void main (String[] args)
    {
        Prueba pru = new Prueba(true,1,'c',"string");
        System.out.println("Boolean = " + pru.b);
        System.out.println("Numerico = " + pru.i);
        System.out.println("Caracter = " + pru.c);
        System.out.println("String = " + pru.s);
    }
}
```

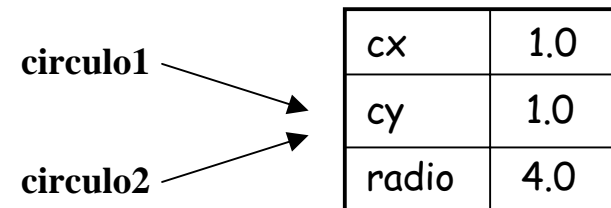
1. Se reserva espacio para el objeto *pru*.
2. Se inicializan las variables miembro a los valores por defecto.
3. Se llama al constructor.

Las variables miembro quedan inicializadas a los valores pasados al constructor



- Constructor de copia (i)
  - En Java el trabajo con objetos es siempre mediante **referencias**.
  - En la primera sentencia del código de la figura, se crea el objeto *circulo1* de la clase *Circulo*, y en la segunda sentencia se declara una nueva referencia, *circulo2*, y se le asigna el valor de la referencia *circulo1*. No se ha creado un nuevo objeto, ambas referencias, *circulo1* y *circulo2* apuntan al mismo objeto (el que se creó en la primera sentencia). En conclusión, **el operador de asignación no sirve para copiar un objeto en otro.**

```
Circulo circulo1 = new Circulo (1.0, 1.0, 4.0);  
Circulo circulo2 = circulo1;
```



- Existen varias soluciones para la copia de objetos:
  - Incorporar un método a la clase que sirva para copiar objetos.
  - Incorporar un constructor de copia, que es un constructor que se invoca para iniciar un objeto creado, a partir de otro existente.



- Constructor de copia (ii)
  - Se define el constructor de copia de la clase *Circulo* de la siguiente manera:

```
public Circulo (Circulo circuloOrigen)
{
    cx = circuloOrigen.cx;
    cy = circuloOrigen.cy;
    radio = circuloOrigen.radio;
}
```

- Se pueden crear nuevos objetos utilizando el constructor de copia:

```
Circulo circulo1 = new Circulo (1.0, 1.0, 4.0);
Circulo circulo2 = new Circulo(circulo1);
```

**circulo1** →

cx	1.0
cy	1.0
radio	4.0

**circulo2** →

cx	1.0
cy	1.0
radio	4.0

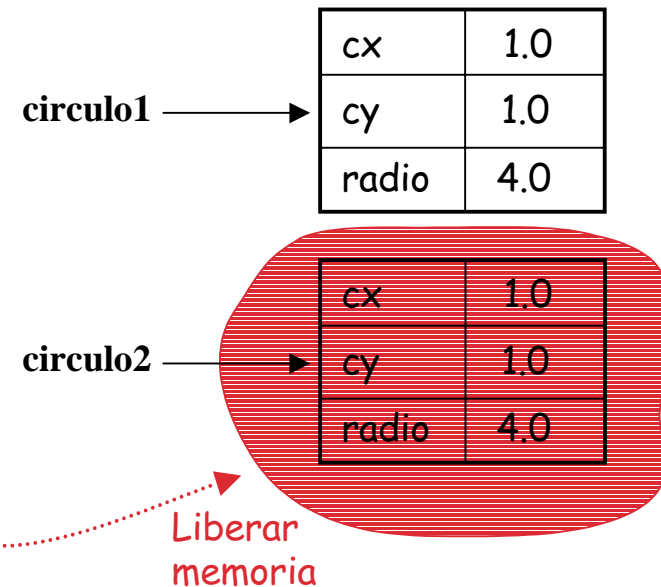
- Concepto
  - Un inicializador estático es un bloque de código sin nombre, sin argumentos y precedido por la palabra reservada **static** que se ejecuta automáticamente al crear la clase (al utilizarla por primera vez).
  - En una clase pueden definirse varios inicializadores estáticos, que se ejecutan en el orden en que sean declarados.
  - Un inicializador estático no puede contener referencias a variables de instancia.
- Ejemplo

```
public class K
{
    static int a = 1;
    static boolean b;
    static double d;
    static
    {
        b = true;
        d = 1.0;
    }
}
```

→ Inicializador estático

- Destrucción de objetos en Java (i)
  - Un objeto es destruido automáticamente (liberada la memoria que ocupa) cuando se eliminan todas las referencias al mismo.
  - Una referencia a un objeto es eliminada cuando:
    - el flujo del programa sale del ámbito donde se ha declarado la referencia.

```
...  
Circulo circulo1 = new Circulo (1.0, 1.0, 4.0);  
{ → Empieza un bloque de código  
    ...  
    Circulo circulo2 = new Circulo(circulo1);  
    ...  
}  
...
```



# Destrucción de objetos en Java (II)

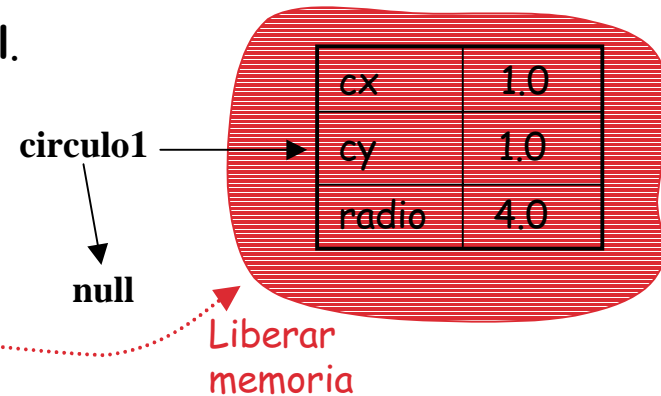


## CLASES

- Destrucción de objetos en Java (ii)

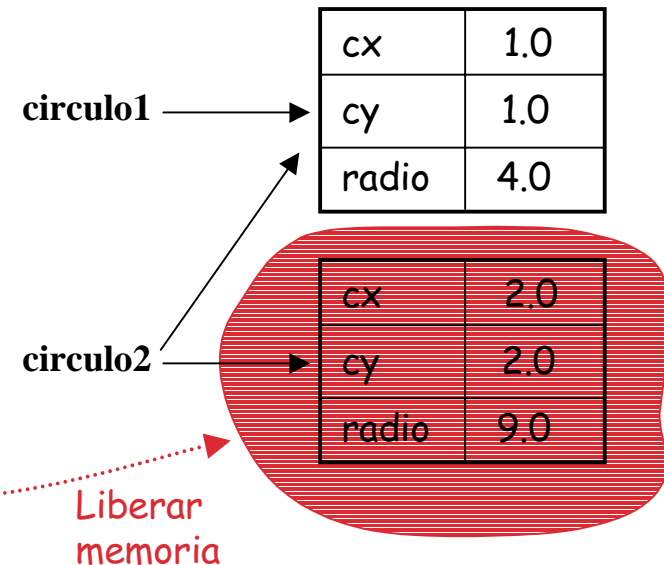
- se le asigna explícitamente el valor `null`.

```
...  
Circulo circulo1 = new Circulo (1.0, 1.0, 4.0);  
...  
circulo1 = null;  
...
```



- se le asigna la dirección de otro objeto.

```
...  
Circulo circulo1 = new Circulo (1.0, 1.0, 4.0);  
Circulo circulo2 = new Circulo(2.0, 2.0, 9.0);  
circulo2 = circulo1;  
...
```



- Destrucción de objetos en Java (iii)
  - A esta característica de Java de liberación automática de la memoria se le llama **garbage collection** (recolección de basura) y se ejecuta en un subproceso paralelamente a la aplicación en ejecución. El **garbage collector** (recolector de basura) mantiene internamente un contador de las referencias a un objeto, y el objeto podrá ser borrado cuando el número de referencias al mismo sea cero.
  - En Java no se sabe exactamente cuando se va a activar el **garbage collector**, pero se puede llamar explícitamente con `System.gc()` aunque esto es considerado como una "sugerencia" al recolector de basura.

- Finalizadores (i)
  - Un finalizador es un método que completa la tarea del recolector de basura, ya que se se invoca automáticamente cuando se va a destruir el objeto, antes de que se libere la memoria.
  - Normalmente se utilizan para operaciones de terminación distintas de liberar memoria (cerrar ficheros, cerrar conexiones de red, etc).
  - Un finalizador:
    - no tiene valor de retorno (**void**)
    - no tiene argumentos
    - se llama siempre **finalize**.
    - No se puede sobrecargar (a diferencia de los constructores)
  - Las clases que no definen un finalizador disponen del proporcionado por la clase **Object**, cuya sintaxis es la siguiente:  

```
protected void finalize() throws Throwable { }
```
  - Una clase define su finalizador sobrescribiendo el método **finalize**.
  - Un finalizador debería terminar siempre llamando al finalizador de su superclase.

- Finalizadores (ii)
  - No se sabe exactamente cuando se va a invocar los finalizadores, ya que se invocan antes de destruir cada objeto, y esta destrucción tampoco se conoce a priori el momento en el que se va a producir. Pero se puede solicitar explícitamente la ejecución de los finalizadores de los objetos listos para destruir con el método `System.runFinalization()` aunque esta invocación es considerada como una "sugerencia" al recolector de basura.