

9.C. Conjuntos.

Sitio: [VIRGEN DE LA PAZ](#)
Curso: Programación
Libro: 9.C. Conjuntos.

Imprimido por: Cristian Esteban Gómez
Día: miércoles, 31 de mayo de 2023, 11:20

Tabla de contenidos

1. Conjuntos (I).
2. Conjuntos (II).
3. Conjuntos (III).
4. Conjuntos (IV).
5. Conjuntos (V).

1. Conjuntos (I).

¿Con qué relacionarías los conjuntos? Seguro que con las matemáticas. Los conjuntos son un tipo de colección que **no admite duplicados**, derivados del concepto matemático de conjunto.

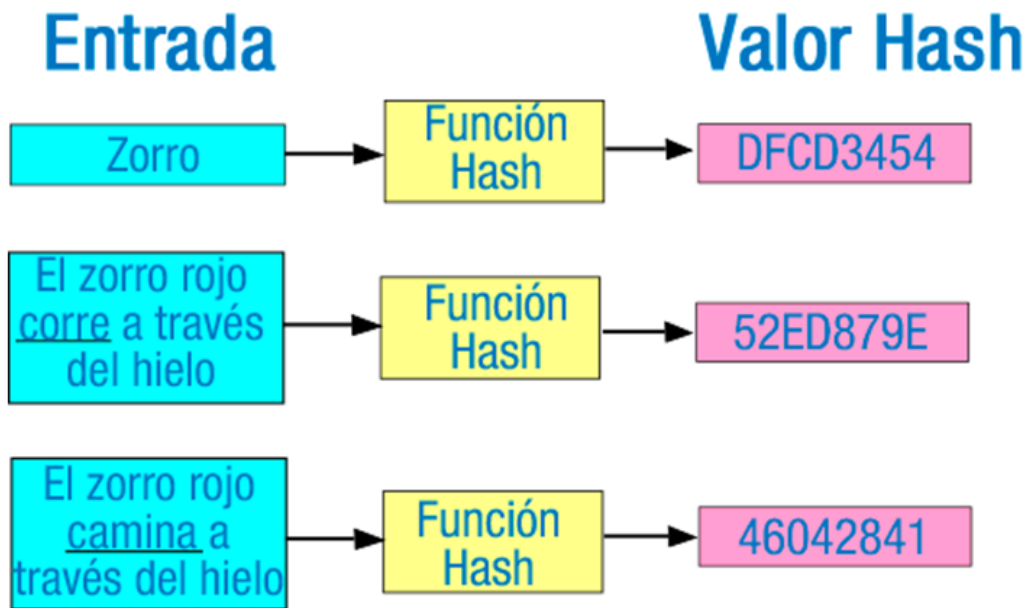


Imagen procedente de curso de Programación MECD.

La interfaz `java.util.Set` define cómo deben ser los conjuntos, y extiende la interfaz `Collection`, aunque no añade ninguna operación nueva. Las implementaciones (clases genéricas que implementan la interfaz `Set`) más usadas son las siguientes:

- `java.util.HashSet`. Conjunto que almacena los objetos usando tablas hash, lo cual acelera enormemente el acceso a los objetos almacenado. Inconvenientes: necesitan bastante memoria y **no almacenan los objetos de forma ordenada** (al contrario pueden aparecer completamente desordenados).
- `java.util.LinkedHashSet`. Conjunto que almacena objetos combinando tablas hash, para un acceso rápido a los datos, y listas enlazadas para conservar el orden. **El orden de almacenamiento es el de inserción**, por lo que se puede decir que es una estructura ordenada a medias. Inconvenientes: necesitan bastante memoria y es algo más lenta que `HashSet`.
- `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores, pero tienen una gran ventaja: **los datos almacenados se ordenan por valor**. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno.

Poco a poco, iremos viendo que son las listas enlazadas y los árboles (no profundizaremos en los árboles rojo-negro, pero si veremos las estructuras tipo árbol en general). Veamos un ejemplo de uso básico de la estructura `HashSet` y después, profundizaremos en los `LinkedHashSet` y los `TreeSet`.

Para crear un conjunto, simplemente creamos el `HashSet` indicando el tipo de objeto que va a almacenar, dado que es una clase genérica que puede trabajar con cualquier tipo de dato debemos crearlo como sigue (no olvides hacer la importación de `java.util.HashSet` primero):

```
HashSet<Integer> conjunto=new HashSet<Integer>();
```

Después podremos ir almacenando objetos dentro del conjunto usando el método `add` (definido por la interfaz `Set`). Los objetos que se pueden insertar serán siempre del tipo especificado al crear el conjunto:

```
Integer n=new Integer(10);
```

```
if (!conjunto.add(n)) System.out.println("Número ya en la lista.");
```

Si el elemento ya está en el conjunto, el método `add` retornará `false` indicando que no se pueden insertar duplicados. Si todo va bien, retornará `true`.

Autoevaluación

¿Cuál de las siguientes estructuras ordena automáticamente los elementos según su valor?

- ☐ `HashSet`.
- ☐ `LinkedHashSet`.
- ☐ `TreeSet`.

2. Conjuntos (II).

Y ahora te preguntará, ¿cómo accedo a los elementos almacenados en un conjunto? Para obtener los elementos almacenados en un conjunto hay que usar iteradores, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (no hay otra forma de acceder a los elementos de un conjunto, es su inconveniente). Los iteradores se ven en mayor profundidad más adelante, de momento, vamos a usar iteradores de forma transparente, a través de una estructura `for` especial, denominada bucle "for-each" o bucle "para cada". En el siguiente código se usa un bucle for-each, en él la variable `i` va tomando todos los valores almacenados en el conjunto hasta que llega al último:

```
for (Integer i: conjunto) {

    System.out.println("Elemento almacenado:"+i);

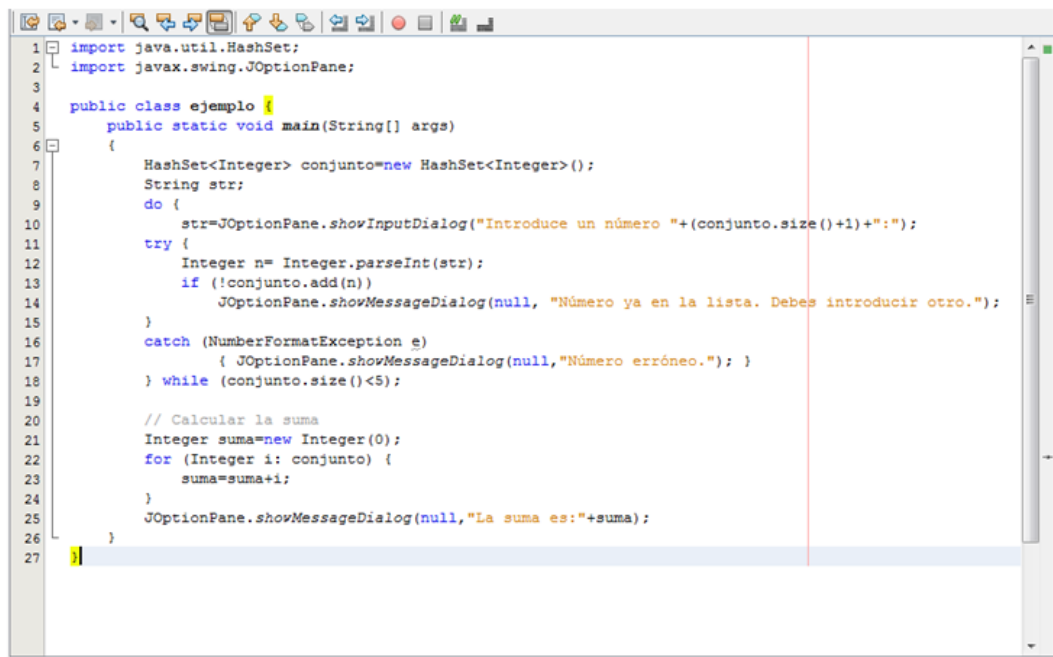
}
```

Como ves la estructura for-each es muy sencilla: la palabra `for` seguida de "(tipo variable:colección)" y el cuerpo del bucle; tipo es el tipo del objeto sobre el que se ha creado la colección, variable pues es la variable donde se almacenará cada elemento de la colección y colección pues la colección en sí. Los bucles for-each se pueden usar para todas las colecciones.

Ejercicio resuelto

Realiza un pequeño programita que pregunte al usuario 5 números diferentes (almacenándolos en un `HashSet`), y que después calcule la suma de los mismos (usando un bucle for-each).

Sol.: Una solución posible podría ser la siguiente. Para preguntar al usuario un número y para mostrarle la información se ha usado la clase `JOptionPane`, pero podrías haber utilizado cualquier otro sistema. Fijate en la solución y verás que el uso de conjuntos ha simplificado enormemente el ejercicio, permitiendo al programador o la programadora centrarse en otros aspectos:



```
1 import java.util.HashSet;
2 import javax.swing.JOptionPane;
3
4 public class ejemplo {
5     public static void main(String[] args)
6     {
7         HashSet<Integer> conjunto=new HashSet<Integer>();
8         String str;
9         do {
10             str=JOptionPane.showInputDialog("Introduce un número "+(conjunto.size()+1)+" :");
11             try {
12                 Integer n= Integer.parseInt(str);
13                 if (!conjunto.add(n))
14                     JOptionPane.showMessageDialog(null, "Número ya en la lista. Debes introducir otro.");
15             }
16             catch (NumberFormatException e)
17             { JOptionPane.showMessageDialog(null,"Número erróneo."); }
18         } while (conjunto.size()<5);
19
20         // Calcular la suma
21         Integer suma=new Integer(0);
22         for (Integer i: conjunto) {
23             suma=suma+i;
24         }
25         JOptionPane.showMessageDialog(null,"La suma es:"+suma);
26     }
27 }
```

```
import java.util.HashSet;
```

```
import javax.swing.JOptionPane;
```

```
public class ejemplo {
```

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

```
{
```

```
    HashSet<Integer> conjunto=new HashSet<Integer>();
```

```
    String str;
```

```
do {  
    str=JOptionPane.showInputDialog("Introduce un número "+(conjunto.size()+1)+" :");  
    try {  
        Integer n= Integer.parseInt(str);  
        if (!conjunto.add(n))  
            JOptionPane.showMessageDialog(null, "Número ya en la lista. Debes introducir otro.");  
    }  
    catch (NumberFormatException e)  
        { JOptionPane.showMessageDialog(null,"Número erróneo."); }  
} while (conjunto.size()<5);  
  
// Calcular la suma  
Integer suma=new Integer(0);  
for (Integer i: conjunto) {  
    suma=suma+i;  
}  
JOptionPane.showMessageDialog(null,"La suma es:"+suma);  
}  
}
```

3. Conjuntos (III).

¿En qué se diferencian las estructuras `LinkedHashSet` y `TreeSet` de la estructura `HashSet`? Ya se comentó antes, y es básicamente en su funcionamiento interno.

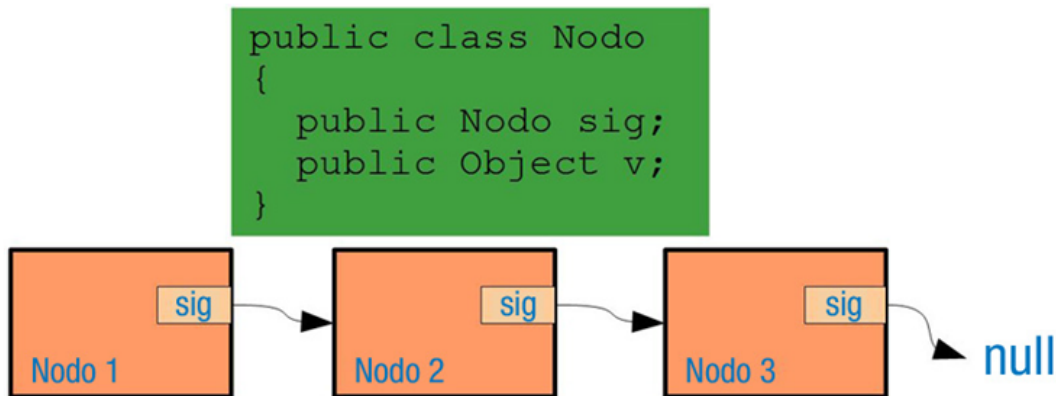


Imagen procedente de curso de Programación del MECD.

La estructura `LinkedHashSet` es una estructura que internamente funciona como una lista enlazada, aunque usa también tablas hash para poder acceder rápidamente a los elementos. Una lista enlazada es una estructura similar a la representada en la imagen de la derecha, la cual está compuesta por nodos (elementos que forman la lista) que van enlazándose entre sí. Un nodo contiene dos cosas: el dato u objeto almacenado en la lista y el siguiente nodo de la lista. Si no hay siguiente nodo, se indica poniendo nulo (`null`) en la variable que contiene el siguiente nodo.

Las listas enlazadas tienen un montón de operaciones asociadas que podremos programar, como eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, ordenación de nodos, etc. Gracias a las colecciones podremos utilizar listas enlazadas sin tener que complicarnos en detalles de programación.

La estructura `TreeSet`, en cambio, utiliza internamente árboles. Los árboles son como las listas pero mucho más complejos. En vez de tener un único elemento siguiente, pueden tener dos o más elementos siguientes, formando estructuras organizadas y jerárquicas.

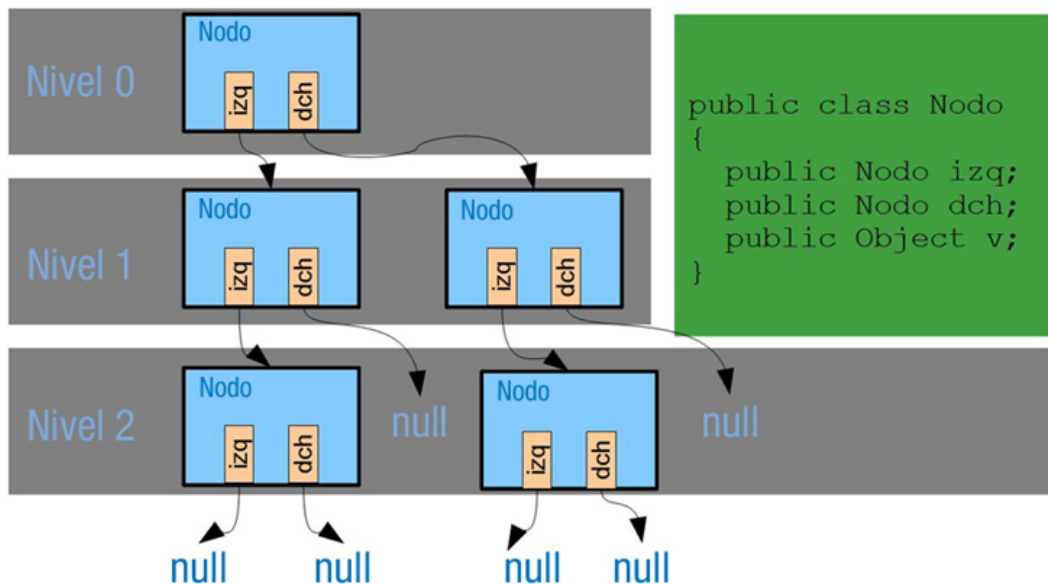


Imagen procedente de curso de Programación del MECD.

Los nodos se diferencian en dos tipos: nodos padre y nodos hijo; un nodo padre puede tener varios nodos hijo asociados (depende del tipo de árbol), dando lugar a una estructura que parece un árbol invertido (de ahí su nombre).

En la figura de la derecha se puede apreciar un árbol donde cada nodo puede tener dos hijos, denominados izquierdo (`izq`) y derecho (`dch`). Puesto que un nodo hijo puede también ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el primer nivel (que no tiene padre).

Los árboles son estructuras complejas de manejar y que permiten operaciones muy sofisticadas. Los árboles usados en los **TreeSet**, los árboles rojo-negro, son árboles auto-ordenados, es decir, que al insertar un elemento, este queda ordenado por su valor de forma que al recorrer el árbol, pasando por todos los nodos, los elementos salen ordenados. El ejemplo mostrado en la imagen es simplemente un árbol binario, el más simple de todos.

[Ver más sobre árboles rojo-negro en Wikipedia.](#)

Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno (inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial. En la siguiente tabla tienes un uso comparado de **TreeSet** y **LinkedHashSet**. Su creación es similar a como se hace con **HashSet**, simplemente sustituyendo el nombre de la clase **HashSet** por una de las otras. Ni **TreeSet**, ni **LinkedHashSet** admiten duplicados, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz **Set** (que es la interfaz que implementan).

Ejemplos de utilización de los conjuntos TreeSet y LinkedHashSet .		
	Conjunto TreeSet .	Conjunto LinkedHashSet .
Ejemplo de uso	<code>TreeSet <Integer> t;</code>	<code>LinkedHashSet <Integer> t;</code>
	<code>t=new TreeSet<Integer>();</code>	<code>t=new LinkedHashSet<Integer>();</code>
	<code>t.add(new Integer(4));</code>	<code>t.add(new Integer(4));</code>
	<code>t.add(new Integer(3));</code>	<code>t.add(new Integer(3));</code>
	<code>t.add(new Integer(1));</code>	<code>t.add(new Integer(1));</code>
	<code>t.add(new Integer(99));</code>	<code>t.add(new Integer(99));</code>
	<code>for (Integer i:t)</code> <code>System.out.println(i);</code>	<code>for (Integer i:t) System.out.println(i);</code>
Resultado mostrado por pantalla	1 3 4 99 (el resultado sale ordenado por valor)	4 3 1 99 (los valores salen ordenados según el momento de inserción en el conjunto)

Autoevaluación

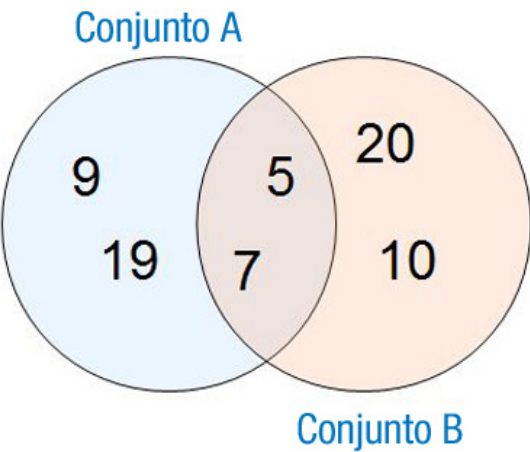
Un árbol cuyos nodos solo pueden tener un único nodo hijo, en realidad es una lista. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

4. Conjuntos (IV).

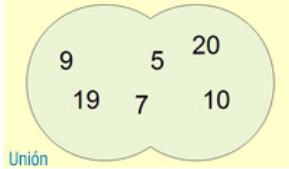
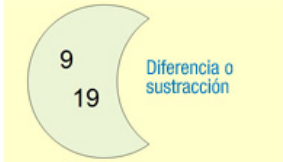
¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle **for** y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se vieron en un apartado anterior, aquí simplemente vamos poner un ejemplo de su uso.


Partimos del siguiente ejemplo, en el que hay dos colecciones de diferente tipo, cada una con 4 números enteros:



```
TreeSet<Integer> A= new TreeSet<Integer>();  
  
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7  
  
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();  
  
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio **Integer** sin tener que hacer nada, lo cual es una ventaja. Veamos las formas de combinar ambas colecciones:

Tipos de combinaciones.		
Combinación.	Código.	Elementos finales del conjunto A.
Unión. Añadir todos los elementos del conjunto B en el conjunto A.	<code>A.addAll(B)</code>	Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20. 
Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.	<code>A.removeAll(B)</code>	Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19. 

Tipos de combinaciones.		
Combinación.	Código.	Elementos finales del conjunto A.
Intersección. Retiene los elementos comunes a ambos conjuntos.	<code>A.retainAll(B)</code>	Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7. 

Recuerda, estas operaciones son comunes a todas las colecciones.

Para saber más

Puede que no recuerdes cómo era eso de los conjuntos, y dada la íntima relación de las colecciones con el álgebra de conjuntos, es recomendable que repases cómo era aquello, con el siguiente artículo de la Wikipedia.

[Álgebra de conjuntos.](#)

Autoevaluación

Tienes un `HashSet` llamado `vocales` que contiene los elementos "a", "e", "i", "o", "u", y otro, llamado `vocales_fuertes` con los elementos "a", "e" y "o". ¿De qué forma podríamos sacar una lista con las denominadas vocales débiles (que son aquellas que no son fuertes)?

- ☐ `vocales.retainAll (vocales_fuertes);`
- ☐ `vocales.removeAll(vocales_fuertes);`
- ☐ No es posible hacer esto con `HashSet`, solo se puede hacer con `TreeSet` o `LinkedHashSet`.

5. Conjuntos (V).

Por defecto, los `TreeSet` ordenan sus elementos de forma ascendente, pero, ¿se podría cambiar el orden de ordenación? Los `TreeSet` tienen un conjunto de operaciones adicionales, además de las que incluye por el hecho de ser un conjunto, que permite entre otras cosas, cambiar la forma de ordenar los elementos. Esto es especialmente útil cuando el tipo de objeto que se almacena no es un simple número, sino algo más complejo (un artículo por ejemplo). `TreeSet` es capaz de ordenar tipos básicos (números, cadenas y fechas) pero otro tipo de objetos no puede ordenarlos con tanta facilidad.

Para indicar a un `TreeSet` cómo tiene que ordenar los elementos, debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello, utilizamos la interfaz genérica `java.util.Comparator`, usada en general en algoritmos de ordenación, como veremos más adelante. Se trata de crear una clase que implemente dicha interfaz, así de fácil. Dicha interfaz requiere de un único método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo. Veamos un ejemplo general de cómo implementar un comparador para una hipotética clase "Objeto":

```
class ComparadorDeObjetos implements Comparator<Objeto> {

    public int compare(Objeto o1, Objeto o2) { ... }

}
```

La interfaz `Comparator` obliga a implementar un único método, es el método `compare`, el cual tiene dos parámetros: los dos elementos a comparar. Las reglas son sencillas, a la hora de personalizar dicho método:

- Si el primer objeto (`o1`) es menor que el segundo (`o2`), debe retornar un número entero negativo.
- Si el primer objeto (`o1`) es mayor que el segundo (`o2`), debe retornar un número entero positivo.
- Si ambos son iguales, debe retornar 0.

A veces, cuando el orden que deben tener los elementos es diferente al orden real (por ejemplo cuando ordenamos los números en orden inverso), la definición de antes puede ser un poco lisa, así que es recomendable en tales casos pensar de la siguiente forma:

- Si el primer objeto (`o1`) debe ir antes que el segundo objeto (`o2`), retornar entero negativo.
- Si el primer objeto (`o1`) debe ir después que el segundo objeto (`o2`), retornar entero positivo.
- Si ambos son iguales, debe retornar 0.

Una vez creado el comparador simplemente tenemos que pasarlo como parámetro en el momento de la creación al `TreeSet`, y los datos internamente mantendrán dicha ordenación:

```
TreeSet<Objeto> ts=new TreeSet<Objeto>(new ComparadorDeObjetos());
```

Ejercicio resuelto

¿Fácil no? Pongámoslo en práctica. Imagínate que Objeto es una clase como la siguiente:

```
class Objeto {

    public int a;

    public int b;

}
```

Imagina que ahora, al añadirlos en un `TreeSet`, estos se tienen que ordenar de forma que la suma de sus atributos (`a` y `b`) sea descendente, ¿cómo sería el comparador?

Solución:

Una de las posibles soluciones a este problema podría ser la siguiente:

```
class ComparadorDeObjetos implements Comparador<Objeto> {

    @Override

    public int compare(Objeto o1, Objeto o2) {
```

```
int sumao1=o1.a+o1.b; int sumao2=o2.a+o2.b;  
  
if (sumao1<sumao2) return 1;  
  
else if (sumao1>sumao2) return -1;  
  
else return 0;  
  
}  
}
```