

TEMA 10: Colecciones de datos.

10.A. Introducción a las estructuras de almacenamiento.

10B. Colecciones

10.C. Conjuntos

10.D. Listas

10.E. Conjuntos de pares clave/valor

10F. Iteradores

10G. Algoritmos sobre listas y arrays

10.A. Introducción a las estructuras de almacenamiento.

1. Introducción.

Anteriormente, en este curso, ya clasificamos las estructuras de almacenamiento en función de si pueden o no cambiar de tamaño de forma dinámica:

- **Estructuras ESTATICAS**, cuyo tamaño se establece en el momento de la creación o definición y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multidimensionales). Este grupo ya lo vimos en una unidad anterior.
- **Estructuras DINAMICAS**, cuyo tamaño es variable (conocidas como **estructuras dinámicas**). Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las listas, árboles, conjuntos y algunos tipos de cadenas de caracteres. Este grupo es el que trataremos en esta unidad de contenidos.

Además, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, ya diferenciamos varios tipos de estructuras:

- **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays. Este grupo ya lo vimos en una unidad anterior.
- **Estructuras ordenadas**. Se trata de estructuras que al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc. Este grupo es el que trataremos en esta unidad de contenidos.

2. Clases y métodos genéricos (I).

Una particularidad esencial de las estructuras dinámicas es la utilización de lo que se conoce como clases y métodos genéricos.

¿Sabes por qué se suele aprender el uso de los genéricos? Pues porque se necesita para usar las listas, aunque realmente los genéricos son una herramienta muy potente y que nos puede ahorrar tareas de programación repetitivas.

Las clases y los métodos genéricos son un recurso de programación disponible en muchos lenguajes de programación. Su objetivo es claro: facilitar la reutilización del software, creando métodos y clases que puedan trabajar con

diferentes tipos de objetos, evitando incómodas y engorrosas conversiones de tipos. Su inicio se remonta a las plantillas (templates) de C++, un gran avance en el mundo de programación sin duda. En lenguajes de más alto nivel como Java o C# se ha transformado en lo que se denomina "genéricos". Veamos un ejemplo sencillo de como transformar un método normal en genérico:

| Versiones genérica y no genérica del método de compararTamano. | |
|--|--|
| Versión no genérica | Versión genérica del método |
| <pre>public class util { public static int compararTamano(Object[] a,Object[] b) { return a.length-b.length; } }</pre> | <pre>public class util { public static <T> int compararTamano (T[] a, T[] b) { return a.length-b.length; } }</pre> |

Los dos métodos anteriores tienen un claro objetivo: permitir comprobar si un array es mayor que otro. Retornarán 0 si ambos arrays son iguales, un número mayor de cero si el array b es mayor, y un número menor de cero si el array a es mayor, pero uno es genérico y el otro no. La versión genérica del módulo incluye la expresión "<T>", justo antes del tipo retornado por el método. "<T>" es la definición de una **variable o parámetro formal de tipo** de la clase o método genérico, al que podemos llamar simplemente **parámetro de tipo o parámetro genérico**. Este parámetro genérico (T) se puede usar a lo largo de todo el método o clase, dependiendo del ámbito de definición, y hará referencia a cualquier clase con la que nuestro algoritmo tenga que trabajar. Como veremos más adelante, puede haber más de un parámetro genérico.

Utilizar genéricos tiene claras ventajas. Para invocar un método genérico, sólo hay que realizar una **invocación de tipo genérico**, olvidándonos de las conversiones de tipo. Esto consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico ("<T>"), para después, pasándole los argumentos correspondientes, ejecutar el algoritmo. Cada clase o interfaz concreta, la podemos denominar **tipo** o **tipo base** y se da por sentado que **los argumentos pasados al método genérico serán también de dicho tipo base**.

Supongamos que el tipo base es Integer, pues para realizar la invocación del método genérico anterior basta con indicar el tipo, entre los símbolos de menor qué y mayor qué ("<Integer>"), justo antes del nombre del método.

| Invocaciones de las versiones genéricas y no genéricas de un método. | |
|--|---|
| Invocación del método no genérico. | Invocación del método genérico. |
| <code>Integer []a={0,1,2,3,4};</code> | <code>Integer []a={0,1,2,3,4};</code> |
| <code>Integer []b={0,1,2,3,4,5};</code> | <code>Integer []b={0,1,2,3,4,5};</code> |
| <code>util.compararTamano ((Object[])a, (Object[])b);</code> | <code>util.<Integer>compararTamano (a, b);</code> |

3. Clases y métodos genéricos (II).

¿Crees qué el código es más legible al utilizar genéricos o que se complica? La verdad es que al principio cuesta, pero después, el código se entiende mejor que si se empieza a insertar conversiones de tipo.

Las clases genéricas son equivalentes a los métodos genéricos pero a nivel de clase, permiten definir un parámetro de tipo o genérico que se podrá usar a lo largo de toda la clase, facilitando así crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase:

```
public class Util<T> {
    T t1;
    public void invertir(T[] array) {
        for (int i = 0; i < array.length / 2; i++) {
            t1 = array[i];
            array[i] = array[array.length - i - 1];
            array[array.length - i - 1] = t1;
        }
    }
}
```

En el ejemplo anterior, la clase Util contiene el método invertir cuya función es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre los símbolos menor que (" $<$ ") y mayor que (" $>$ "), justo detrás del nombre de la clase. Veamos un ejemplo:

```
Integer[] numeros={0,1,2,3,4,5,6,7,8,9};
Util<Integer> u= new Util<Integer>();
u.invertir(numeros);
for (int i=0;i<numeros.length;i++)

    System.out.println(numeros[i]);
```

Como puedes observar, el uso de genéricos es sencillo, tanto a nivel de clase como a nivel de método. Simplemente, a la hora de crear una instancia de una clase genérica, hay que especificar el tipo, tanto en la definición (Util <integer> u) como en la creación (new Util<Integer>()).

Los genéricos los vamos a usar ampliamente a partir de ahora, aplicados a un montón de clases genéricas que tiene Java y que son de gran utilidad, por lo que es conveniente que aprendas bien a usar una clase genérica.

Los parámetros de tipo de las clases genéricas solo pueden ser clases, no pueden ser jamás tipos de datos primitivos como `int`, `short`, `double`, etc. En su lugar, debemos usar sus clases envoltorio `Integer`, `Short`, `Double`, etc.

10B. Colecciones

1. Introducción a las colecciones.

¿Qué consideras una colección? Pues seguramente al pensar en el término se te viene a la cabeza una colección de libros o algo parecido, y la idea no va muy desencaminada. **Una colección a nivel de software es un grupo de elementos almacenados de forma conjunta en una misma estructura.** Eso son las colecciones.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permiten manejar grupos de objetos, todo ello enfocado a potenciar la reusabilidad del software y facilitar las tareas de programación. Te parecerá increíble el tiempo que se ahorra empleando colecciones y cómo se reduce la complejidad del software usándolas adecuadamente. Las colecciones permiten almacenar y manipular grupos de objetos que, a priori, están relacionados entre sí (aunque no es obligatorio que estén relacionados, lo lógico es que si se almacenan juntos es porque tienen alguna relación entre sí), pudiendo trabajar con cualquier tipo de objeto (de ahí que se empleen los genéricos en las colecciones).

Además, las colecciones permiten realizar algunas operaciones útiles sobre los elementos almacenados, tales como búsqueda u ordenación. En algunos casos es necesario que los objetos almacenados cumplan algunas condiciones (que implementen algunas interfaces), para poder ser hacer uso de estos algoritmos.

Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación. En algunos lenguajes de programación su uso es algo más complejo (como es el caso de C++), pero en Java su uso es bastante sencillo, es algo que descubrirás a lo largo de lo que queda de tema.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un modelo de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados, por lo que es necesario conocerlas. La interfaz inicial, a través de la cual se han construido el resto de colecciones, es la interfaz `java.util.Collection`, que define las operaciones comunes a todas las

colecciones derivadas. A continuación se muestran las operaciones más importantes definidas por esta interfaz, ten en cuenta que `Collection` es una interfaz genérica donde "`<E>`" es el parámetro de tipo (podría ser cualquier clase):

- Método `int size()`: retorna el número de elementos de la colección.
- Método `boolean isEmpty()`: retornará verdadero si la colección está vacía.
- Método `boolean contains (Object element)`: retornará verdadero si la colección tiene el elemento pasado como parámetro.
- Método `boolean add(E element)`: permitirá añadir elementos a la colección.
- Método `boolean remove (Object element)`: permitirá eliminar elementos de la colección.
- Método `Iterator<E> iterator()`: permitirá crear un iterador para recorrer los elementos de la colección. Esto se ve más adelante, no te preocupes.
- Método `Object[] toArray()`: permite pasar la colección a un array de objetos tipo `Object`.
- Método `containsAll(Collection<?> c)`: permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- Método `addAll (Collection<? extends E> c)`: permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
- Método `boolean removeAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- Método `boolean retainAll(Collection<?> c)`: si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
- Método `void clear()`: vacía la colección.

Más adelante veremos cómo se usan estos métodos, será cuando veamos las implementaciones (clases genéricas que implementan alguna de las interfaces derivadas de la interfaz `Collection`).

10.C. Conjuntos

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.

Entrada

Valor Hash

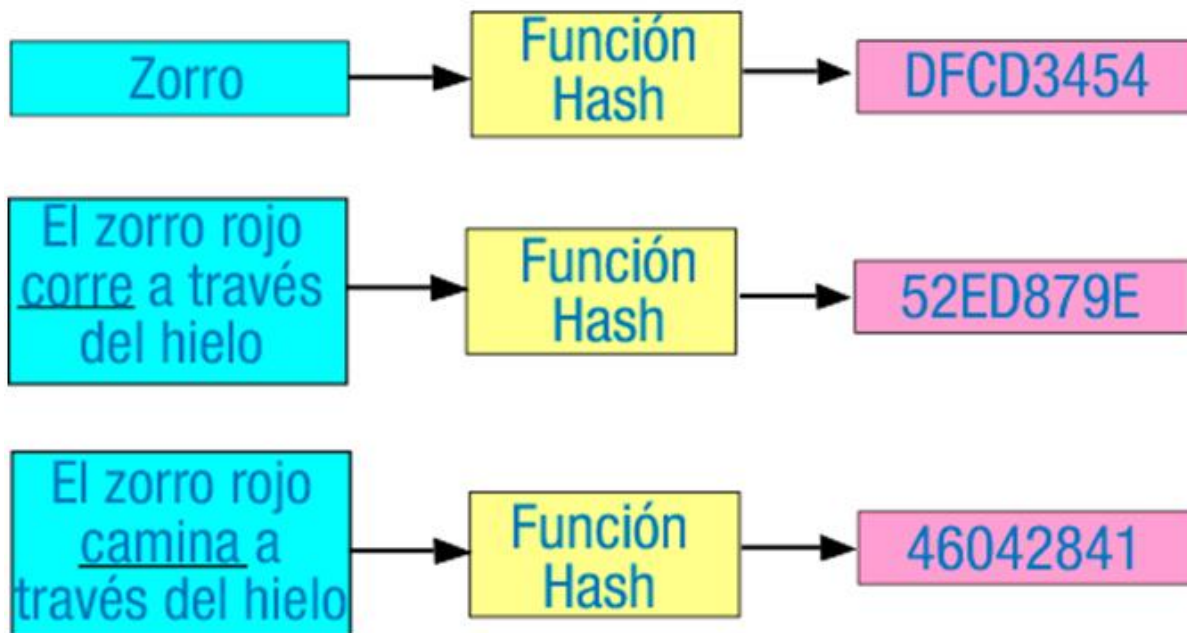


Imagen procedente de curso de Programación MECD.

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.

2. Conjuntos (II).

Y ahora te preguntarás, ¿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:

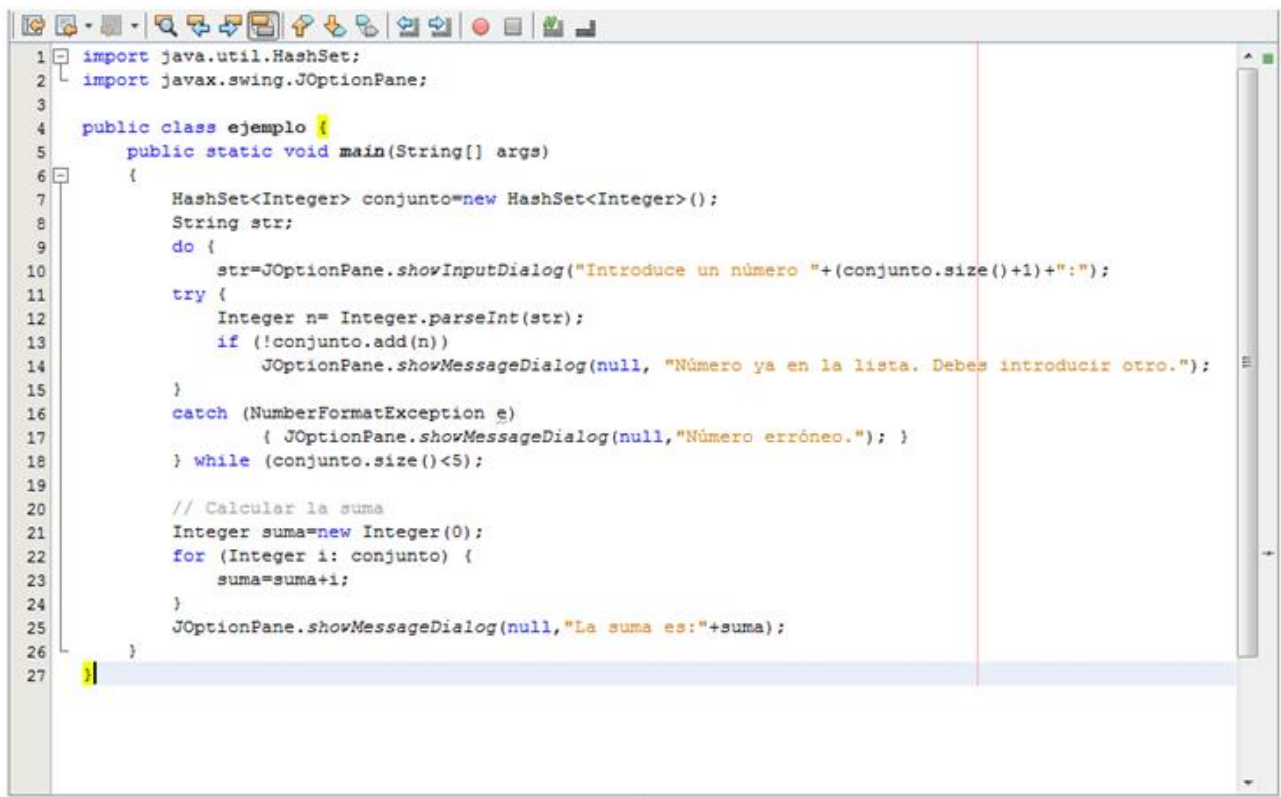
```
for (Integer i: conjunto) {  
    System.out.println("Elemento almacenado:"+i);  
}
```

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. Fíjate en la solución y verás que el uso de conjuntos ha si

A screenshot of a Java IDE window showing a code file. The code is a Java class named 'ejemplo' with a static method 'main'. It uses 'HashSet' from 'java.util' and 'JOptionPane' from 'javax.swing'. The code prompts the user to enter a number, adds it to a HashSet if it's not already there, and shows error messages for invalid input or duplicates. It also calculates the sum of the numbers in the set. The code is as follows:

```
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);

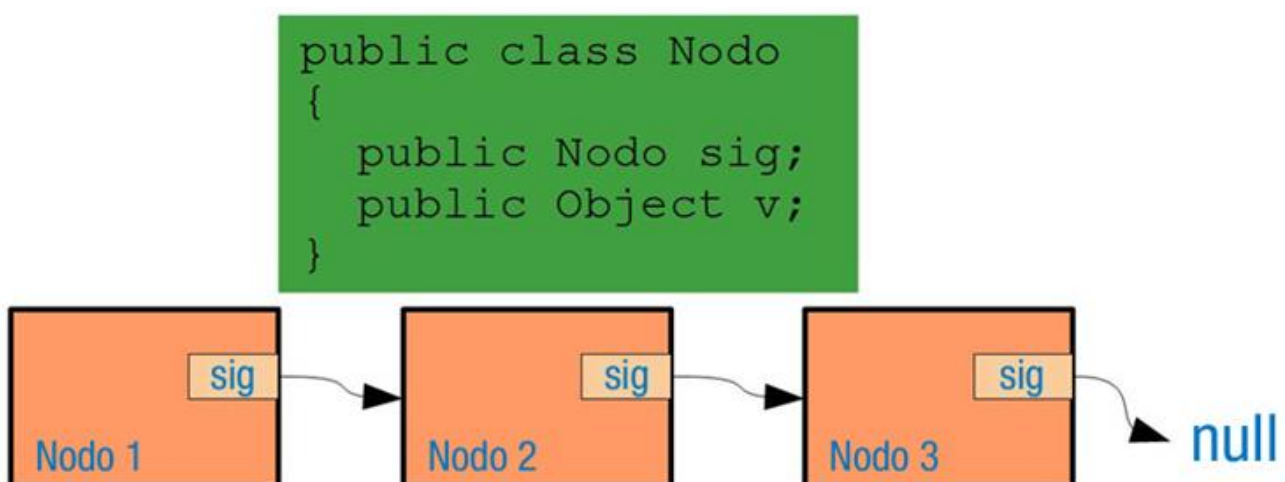
}

}

```

3. Conjuntos (III).

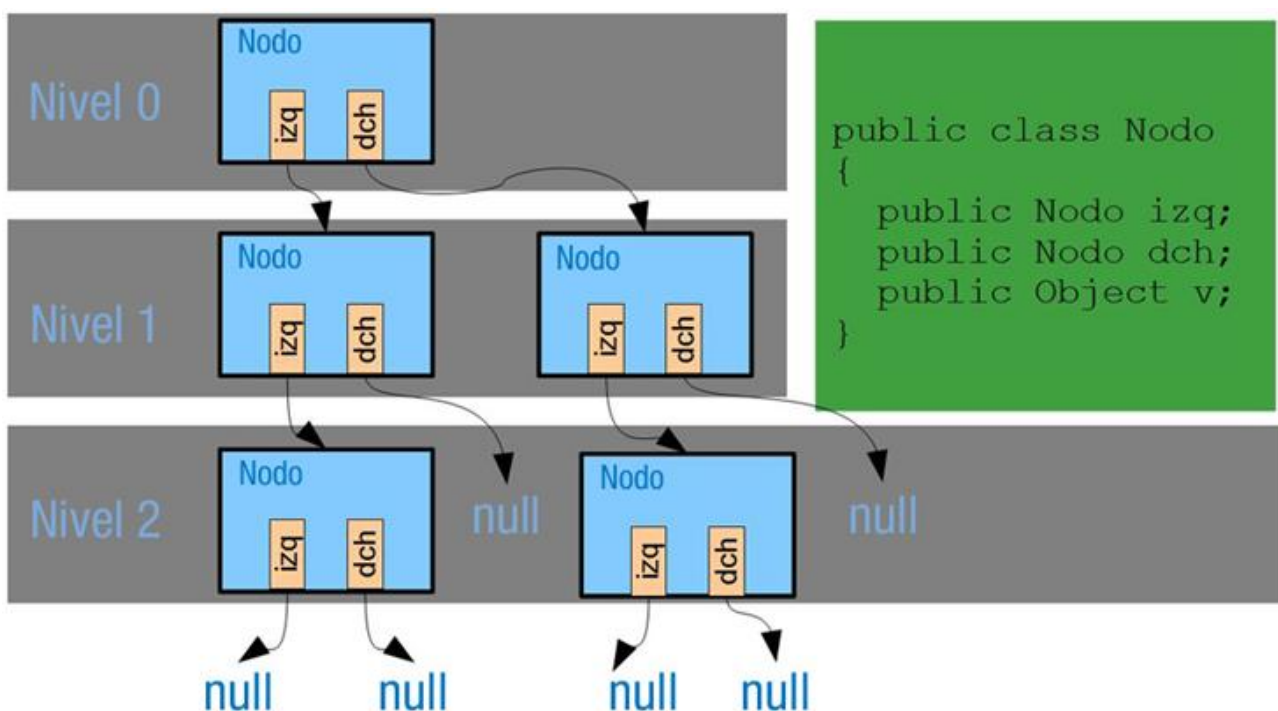
¿En qué se diferencian las estructuras LinkedHashSet y TreeSet de la estructura HashSet? Ya se comento antes, y es básicamente en su funcionamiento interno.



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.



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.](#)

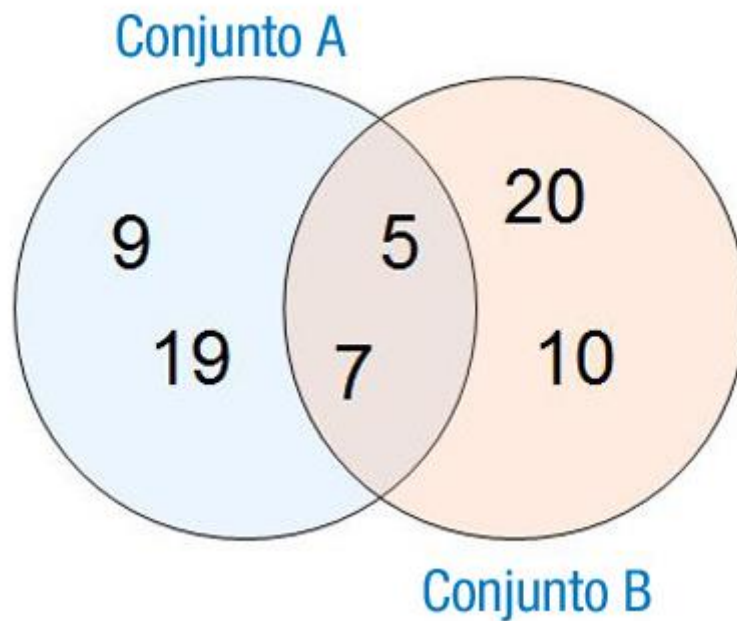
Nuevamente, no se va a profundizar en las operaciones que se pueden realizar en un árbol a nivel interno (inserción de nodos, eliminación de nodos, búsqueda de un valor, etc.). Nos aprovecharemos de las colecciones para hacer uso de su potencial. En la siguiente tabla tienes un uso comparado de TreeSet 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 | <pre>TreeSet <Integer> t; t=new TreeSet<Integer>(); t.add(new Integer(4)); t.add(new Integer(3)); t.add(new Integer(1)); t.add(new Integer(99)); for (Integer i:t) System.out.println(i);</pre> | <pre>LinkedHashSet <Integer> t; t=new LinkedHashSet<Integer>(); t.add(new Integer(4)); t.add(new Integer(3)); t.add(new Integer(1)); t.add(new Integer(99)); for (Integer i:t) System.out.println(i);</pre> |
| Resultado mostrado por pantalla | <pre>1 3 4 99</pre> <p>(el resultado sale ordenado por valor)</p> | <pre>4 3 1 99</pre> <p>(los valores salen ordenados según el momento de inserción en el conjunto)</p> |

4. Conjuntos (IV).

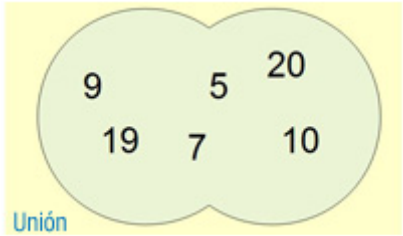
¿Cómo podría copiar los elementos de un conjunto de uno a otro? ¿Hay que usar un bucle for y recorrer toda la lista para ello? ¡Qué va! Para facilitar esta tarea, los conjuntos, y las colecciones en general, facilitan un montón de operaciones para poder combinar los datos de varias colecciones. Ya se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

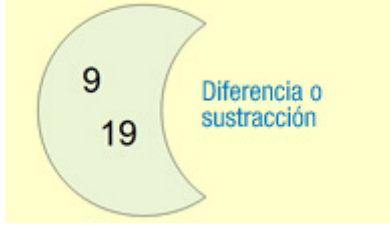

Partimos del siguiente ejemplo, en el que hay dos colecciones de diferente tipo, cada una con 4 números enteros:



```
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
    HashSet<Integer> B= new HashSet<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. | A.addAll(B) | <p>Todos los del conjunto A, añadiendo los del B, pero sin repetir los que ya están: 5, 7, 9, 10, 19 y 20.</p>  <p>Unión</p> |
| Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A. | A.removeAll(B) | <p>Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.</p> |

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

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.](#)

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 liosa, 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;  
    }  
}
```

10.D. Listas

1. Listas (I).

¿En qué se diferencia una lista de un conjunto? Las listas son elementos de programación un poco más avanzados que los conjuntos. Su ventaja es que amplían el conjunto de operaciones de las colecciones añadiendo operaciones extra, veamos algunas de ellas:

- Las listas sí pueden almacenar duplicados, si no queremos duplicados, hay que verificar manualmente que el elemento no esté en la lista antes de su inserción.
- Acceso posicional. Podemos acceder a un elemento indicando su posición en la lista.
- Búsqueda. Es posible buscar elementos en la lista y obtener su posición. En los conjuntos, al ser colecciones sin aportar nada nuevo, solo se podía comprobar si un conjunto contenía o no un elemento de la lista, retornando verdadero o falso. Las listas mejoran este aspecto.
- Extracción de sublistas. Es posible obtener una lista que contenga solo una parte de los elementos de forma muy sencilla.

En Java, para las listas se dispone de una interfaz llamada `java.util.List`, y dos implementaciones (`java.util.LinkedList` y `java.util.ArrayList`), con diferencias significativas entre ellas. Los métodos de la interfaz `List`, que obviamente estarán en todas las implementaciones, y que permiten las operaciones anteriores son:

- `E get(int index)`. El método `get` permite obtener un elemento partiendo de su posición (`index`).
- `E set(int index, E element)`. El método `set` permite cambiar el elemento almacenado en una posición de la lista (`index`), por otro (`element`).

- `void add(int index, E element)`. Se añade otra versión del método `add`, en la cual se puede insertar un elemento (`element`) en la lista en una posición concreta (`index`), desplazando los existentes.
- `E remove(int index)`. Se añade otra versión del método `remove`, esta versión permite eliminar un elemento indicando su posición en la lista.
- `boolean addAll(int index, Collection<? extends E> c)`. Se añade otra versión del método `addAll`, que permite insertar una colección pasada por parámetro en una posición de la lista, desplazando el resto de elementos.
- `int indexOf(Object o)`. El método `indexOf` permite conocer la posición (índice) de un elemento, si dicho elemento no está en la lista retornará -1.
- `int lastIndexOf(Object o)`. El método `lastIndexOf` nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista si puede almacenar duplicados).
- `List<E> subList(int from, int to)`. El método `subList` genera una sublista (una vista parcial de la lista) con los elementos comprendidos entre la posición inicial (incluida) y la posición final (no incluida).

Ten en cuenta que los elementos de una lista empiezan a numerarse por 0. Es decir, que el primer elemento de la lista es el 0. Ten en cuenta también que `List` es una interfaz genérica, por lo que `<E>` corresponde con el tipo base usado como parámetro genérico al crear la lista.

10.E. Conjuntos de pares clave/valor

1. Conjuntos de pares clave/valor.

¿Cómo almacenarías los datos de un diccionario? Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existen precisamente los arrays asociativos. Un tipo de array asociativo son los mapas o diccionarios, que permiten almacenar pares de valores conocidos como clave y valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En Java existe la interfaz `java.util.Map` que define los métodos que deben tener los mapas, y existen tres implementaciones principales de dicha interfaz: `java.util.HashMap`, `java.util.TreeMap` y `java.util.LinkedHashMap`. ¿Te suenan? Claro que sí. Cada una de ellas, respectivamente, tiene características similares a `HashSet`, `TreeSet` y `LinkedHashSet`, tanto en funcionamiento interno como en rendimiento.

Los **mapas utilizan clases genéricas** para dar extensibilidad y flexibilidad, y permiten definir un tipo base para la clave, y otro tipo diferente para el valor.

Veamos un ejemplo de cómo crear un mapa, que es extensible a los otros dos tipos de mapas:

```
HashMap<String,Integer> t=new HashMap<String,Integer>();
```

El mapa anterior permite usar cadenas como llaves y almacenar de forma asociada a cada llave, un número entero. Veamos los métodos principales de la interfaz Map, disponibles en todas las implementaciones. En los ejemplos, *v* es el tipo base usado para el valor y *k* el tipo base usado para la llave:

| Métodos principales de los mapas. | |
|---|--|
| Método. | Descripción. |
| V put(K key, V value); | Inserta un par de objetos llave (<i>key</i>) y valor (<i>value</i>) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará <i>null</i> . |
| V get(Object key); | Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará <i>null</i> . |
| V remove(Object key); | Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o <i>null</i> , si la llave no existe. |
| boolean containsKey(Object key); | Retornará <i>true</i> si el mapa tiene almacenada la llave pasada por parámetro, <i>false</i> en cualquier otro caso. |
| boolean containsValue(Object value); | Retornará <i>true</i> si el mapa tiene almacenado el valor pasado por parámetro, <i>false</i> en cualquier otro caso. |
| int size(); | Retornará el número de pares llave y valor almacenado en el mapa. |
| boolean isEmpty(); | Retornará <i>true</i> si el mapa está vacío, <i>false</i> en cualquier otro caso. |
| void clear(); | Vacía el mapa. |

10F. Iteradores

1. Iteradores (I).

¿Qué son los iteradores realmente? Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma sencilla, de forma secuencial, y de forma segura. Los mapas, como no derivan de la interfaz Collection realmente, no tienen iteradores, pero como veremos, existe un truco interesante.

Los iteradores permiten recorrer las colecciones de dos formas: **bucles for-each** (existentes en Java a partir de la versión 1.5) **y a través de un bucle normal creando un iterador**. Como los bucles for-each ya los hemos visto antes (y ha quedado patente su simplicidad), nos vamos a centrar en el otro método, especialmente útil en versiones antiguas de Java. Ahora la pregunta es, ¿cómo se crea un iterador? Pues invocando el método "iterator()" de

cualquier colección. Veamos un ejemplo (en el ejemplo t es una colección cualquiera):

```
Iterator<Integer> it=t.iterator();
```

Fijate que se ha especificado un parámetro para el tipo de dato genérico en el iterador (poniendo "<Integer>" después de Iterator). Esto es porque los iteradores son también clases genéricas, y es necesario especificar el tipo base que contendrá el iterador. Sino se especifica el tipo base del iterador, igualmente nos permitiría recorrer la colección, pero retornará objetos tipo Object (clase de la que derivan todas las clases), con lo que nos veremos obligados a forzar la conversión de tipo.

Para recorrer y gestionar la colección, el iterador ofrece tres métodos básicos:

- `boolean hasNext()`. Retornará true si le quedan más elementos a la colección por visitar. False en caso contrario.
- `E next()`. Retornará el siguiente elemento de la colección, si no existe siguiente elemento, lanzará una excepción (`NoSuchElementException` para ser exactos), con lo que conviene chequear primero si el siguiente elemento existe.
- `remove()`. Elimina de la colección el último elemento retornado en la última invocación de `next` (no es necesario pasárselo por parámetro). Cuidado, si `next` no ha sido invocado todavía, saltará una incomoda excepción.

¿Cómo recorreríamos una colección con estos métodos? Pues de una forma muy sencilla, un simple bucle mientras (`while`) con la condición `hasNext()` nos permite hacerlo:

```
while (it.hasNext()) // Mientras que haya un siguiente elemento, seguiremos en el bucle.
{
    Integer t=it.next(); // Escogemos el siguiente elemento.
    if (t%2==0) it.remove(); //Si es necesario, podemos eliminar el elemento extraído de la lista.
}
```

¿Qué elementos contendría la lista después de ejecutar el bucle? Efectivamente, solo los números impares.

Reflexiona

Las listas permiten acceso posicional a través de los métodos `get` y `set`, y acceso secuencial a través de iteradores, ¿cuál es para tí la forma más cómoda de recorrer todos los elementos? ¿Un acceso posicional a través un bucle "`for (i=0;i<lista.size();i++)`" o un acceso secuencial usando un bucle "`while (iterador.hasNext())`"?

2. Iteradores (II).

¿Qué inconvenientes tiene usar los iteradores sin especificar el tipo de objeto? En el siguiente ejemplo, se genera una lista con los números del 0 al 10. De la lista, se eliminan aquellos que son pares y solo se dejan los impares. En el ejemplo de la izquierda se especifica el tipo de objeto del iterador, en el ejemplo de la derecha no, observa el uso de la conversión de tipos en la línea 6.

| Comparación de usos de los iteradores, con o sin conversión de tipos. | |
|--|--|
| Ejemplo indicando el tipo de objeto de iterador | Ejemplo no indicando el tipo de objeto del iterador |
| <pre> ArrayList <Integer> lista=new ArrayList<Integer>(); for (int i=0;i<10;i++) lista.add(i); Iterator<Integer> it=lista.iterator(); while (it.hasNext()) { Integer t=it.next(); if (t%2==0) it.remove(); } </pre> | <pre> ArrayList <Integer> lista=new ArrayList<Integer>(); for (int i=0;i<10;i++) lista.add(i); Iterator it=lista.iterator(); while (it.hasNext()) { Integer t=(Integer)it.next(); if (t%2==0) it.remove(); } </pre> |

Un iterador es seguro porque está pensado para no sobrepasar los límites de la colección, ocultando operaciones más complicadas que pueden repercutir en errores de software. Pero realmente se convierte en inseguro cuando es necesario hacer la operación de conversión de tipos. Si la colección no contiene los objetos esperados, al intentar hacer la conversión, saltará una incómoda excepción. Usar genéricos aporta grandes ventajas, pero usándolos adecuadamente.

Para recorrer los mapas con iteradores, hay que hacer un pequeño truco. Usamos el método `entrySet` que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien, el método `keySet` para generar un conjunto con las llaves existentes en el mapa. Veamos cómo sería para el segundo caso, el más sencillo:

```

HashMap<Integer,Integer> mapa=new HashMap<Integer,Integer>test();
for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.
for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet,
    contendrá las llaves.
{
    Integer valor=mapa.get(llave); //Para cada llave, accedemos a su
    valor si es necesario.
}

```

Lo único que tienes que tener en cuenta es que el conjunto generado por **keySet** no tendrá obviamente el método `add` para añadir elementos al mismo, dado que eso tendrás que hacerlo a través del mapa.

Recomendación

Si usas iteradores, y piensas eliminar elementos de la colección (e incluso de un mapa), debes usar el método `remove` del iterador y no el de la colección. Si eliminas los elementos utilizando el método `remove` de la colección, mientras estás dentro de un bucle de iteración, o dentro de un bucle `for-each`, los fallos que pueden producirse en tu programa son impredecibles. ¿Logras adivinar porqué se pueden producir dichos problemas?

Los problemas son debidos a que el método `remove` del iterador elimina el elemento de dos sitios: de la colección y del iterador en sí (que mantiene interiormente información del orden de los elementos). Si usas el método `remove` de la colección, la información solo se elimina de un lugar, de la colección.

10G. Algoritmos sobre listas y arrays

1. Algoritmos (I).

La palabra algoritmo seguro que te suena, pero, ¿a qué se refiere en el contexto de las colecciones y de otras estructuras de datos? Las colecciones, los arrays e incluso las cadenas, tienen un conjunto de operaciones típicas asociadas que son habituales. Algunas de estas operaciones ya las hemos visto antes, pero otras no. Veamos para qué nos pueden servir estas operaciones:

- Ordenar listas y arrays.
- Desordenar listas y arrays.
- Búsqueda binaria en listas y arrays.
- Conversión de arrays a listas y de listas a array.
- Partir cadenas y almacenar el resultado en un array.

Estos algoritmos están en su mayoría recogidos como métodos estáticos de las clases `java.util.Collections` y `java.util.Arrays`, salvo los referentes a cadenas obviamente.

Los algoritmos de ordenación ordenan los elementos en orden natural, siempre que Java sepa como ordenarlos. Como se explico en el apartado de conjuntos, cuando se desea que la ordenación siga un orden diferente, o simplemente los elementos no son ordenables de forma natural, hay que facilitar un mecanismo para que se pueda producir la ordenación. Los tipos "ordenables" de forma natural son los enteros, las cadenas (orden alfabético) y las fechas, y por defecto su orden es ascendente.

La clase `Collections` y la clase `Arrays` facilitan el método `sort`, que permiten ordenar respectivamente listas y arrays. Los siguientes ejemplos ordenarían los números de forma ascendente (de menor a mayor):

| |
|---|
| Ordenación natural en listas y arrays. |
|---|

| Ejemplo de ordenación de un array de números | Ejemplo de ordenación de una lista con números |
|---|---|
| <pre>Integer[] array={10,9,99,3,5}; Arrays.sort(array);</pre> | <pre>ArrayList<Integer> lista=new ArrayList<Integer>(); lista.add(10); lista.add(9);lista.add(99); lista.add(3); lista.add(5); Collections.sort(lista);</pre> |

2. Algoritmos (II).

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. Volvamos al ejemplo tratado ya en alguna ocasión en el curso, relativo a un conjunto de artículos y que quisiéramos ordenar por código del artículo. Imagina que tienes los artículos almacenados en una lista llamada "articulos", y que cada artículo se almacena en la siguiente clase (fíjate que el código de artículo es una cadena y no un número):

```
class Artículo {
    public String codArticulo; // Código de artículo
    public String descripcion; // Descripción del artículo.
    public int cantidad; // Cantidad a proveer del artículo.
}
```

La primera forma de ordenar consiste en crear una clase que implemente la interfaz `java.util.Comparator`, y en ende, el método `compare` definido en dicha interfaz. Esto se explicó en el apartado de conjuntos, al explicar el `TreeSet`, así que no vamos a profundizar en ello. No obstante, el comparador para ese caso podría ser así:

```
class comparadorArticulos implements Comparator<Articulo>
{
    @Override
    public int compare( Articulo o1, Articulo o2) { return
o1.codArticulo.compareTo(o2.codArticulo); }
}
```

Una vez creada esta clase, ordenar los elementos es muy sencillo, simplemente se pasa como segundo parámetro del método `sort` una instancia del comparador creado:

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma es quizás más sencilla cuando se trata de objetos cuya ordenación no existe de forma natural, pero requiere modificar la clase `Articulo`. Consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz `java.util.Comparable`. **Todos los objetos que**

implementan la interfaz Comparable son "ordenables" y se puede invocar el método sort sin indicar un comparador para ordenarlos. La interfaz comparable solo requiere implementar el método compareTo:

```
class Artículo implements Comparable<Artículo>{
    public String codArtículo;
    public String descripcion;
    public int cantidad;

    @Override
    public int compareTo(Artículo o) { return codArtículo.compareTo(o.codArtículo); }
}
```

Del ejemplo anterior se pueden denotar dos cosas importantes: que la interfaz Comparable es genérica y que para que funcione sin problemas es conveniente indicar el tipo base sobre el que se permite la comparación (en este caso, el objeto Artículo debe compararse consigo mismo), y que el método compareTo solo admite un parámetro, dado que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método compareTo es el mismo que el método compare de la interfaz Comparator: si la clase que se pasa por parámetro es igual al objeto, se tendría que retornar 0; si es menor o anterior, se debería retornar un número menor que cero; si es mayor o posterior, se debería retornar un número mayor que 0.

Ordenar ahora la lista de artículos es sencillo, fíjate que fácil:
"Collections.sort(articulos);"

3. Algoritmos (III).

¿Qué más ofrece las clases java.util.Collections y java.util.Arrays de Java? Una vez vista la ordenación, quizás lo más complicado, veamos algunas operaciones adicionales. En los ejemplos, la variable "array" es un array y la variable "lista" es una lista de cualquier tipo de elemento:

| Operaciones adicionales sobre listas y arrays. | | |
|--|--|--|
| Operación | Descripción | Ejemplos |
| Desordenar una lista. | Desordena una lista, este método no está disponible para arrays. | <code>Collections.shuffle (lista);</code> |
| Rellenar una lista o array. | Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array. | <code>Collections.fill (lista,elemento);</code> <code>Arrays.fill (array,elemento);</code> |
| Búsqueda binaria. | Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o | <code>Collections.binarySearch(lista,elemento);</code> <code>Arrays.binarySearch(array, elemento);</code> |

| | | |
|-------------------------------------|---|--|
| | array estén ordenados, si no lo están, la búsqueda no tendrá éxito. | |
| Convertir un array a lista. | Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es <code>ArrayList</code> ni <code>LinkedList</code>), solo se especifica que retorna una lista que implementa la interfaz <code>java.util.List</code> . | <pre>List lista=Arrays.asList(array);</pre> <p>Si el tipo de dato almacenado en el array es conocido (<code>Integer</code> por ejemplo), es conveniente especificar el tipo de objeto de la lista:</p> <pre>List<Integer>lista = Arrays.asList(array);</pre> |
| Convertir una lista a array. | Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase <code>Collections</code> , sino propio de la interfaz <code>Collection</code> . Es conveniente que sepas de su existencia. | <p>Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista:</p> <pre>Integer[] array=new Integer[lista.size()]; lista.toArray(array)</pre> |
| Dar la vuelta. | Da la vuelta a una lista, poniéndola en orden inverso al que tiene. | <pre>Collections.reverse(lista);</pre> |

Otra operación que no se ha visto hasta ahora es la dividir una cadena en partes. Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas. Es una operación sencilla, pero dado que es necesario conocer el funcionamiento de los arrays y de las expresiones regulares para su uso, no se ha podido ver hasta ahora. Para poder realizar esta operación, usaremos el método `split` de la clase `String`. El delimitador o separador es una expresión regular, único argumento del método `split`, y puede ser obviamente todo lo complejo que sea necesario:

```
String texto="Z,B,A,X,M,O,P,U";
String []partes=texto.split(",");
Arrays.sort(partes);
```

En el ejemplo anterior la cadena `texto` contiene una serie de letras separadas por comas. La cadena se ha dividido con el método `split`, y se ha guardado cada carácter por separado en un array. Después se ha ordenado el array. ¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!

