**UNIDAD 7: Estructuras de datos internas (memoria)**

**1.- Clases y métodos genéricos (I)**

**Las clases y los métodos genéricos** son un recurso de programación disponible en muchos lenguajes. 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 cómo transformar un método normal en genérico:

| **Versión no genérica** | **Versión genérica del método** |
| --- | --- |
| public class Util {  public static int compararTamano(Object[] a, Object[] b) {  return a.length-b.length;  }  } | public class Util {  public static <T> int compararTamano (T[] a, T[] b) {  return a.length-b.length;  }  } |

Los dos métodos anteriores tienen el mismo objetivo: permitir comprobar si un array es más grande que otro (su longitud es mayor). Retornarán 0 si ambos arrays tienen la misma longitud, un número mayor de cero si el array a es más grande (su longitud es mayor), y un número menor de cero si el array b es más grande. La diferencia está en que un de los métodos es genérico y el otro no. La versión genérica del método 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.

| **Invocación del método no genérico.** | **Invocación del método genérico.** |
| --- | --- |
| Integer[] a = {0,1,2,3,4};  Integer[] b = {0,1,2,3,4,5};  Util.compararTamano ((Object[])a, (Object[])b); | Integer[] a = {0,1,2,3,4};  Integer[] b = {0,1,2,3,4,5};  Util.<Integer>compararTamano (a, b); |

Decir que una clase o un método sea "genérico" significa que está "parametrizada". Esto es útil porque permite crear clases, interfaces y métodos en los que el tipo de datos con el que se trabaja se indica como parámetro. 

La principal ventaja de poder escribir código genérico es que podemos escribir un método (o incluso una clase completa) indicando que un tipo determinado de un atributo o de un parámetro es de tipo <T>, en lugar de que es de tipo *Integer*, o  *Double*, o *LocalDate*, o *String,* o *Dispositivo*. Esto en realidad ya podía hacerse si declaramos un tipo como *Object*, pues cualquier clase Java es de tipo *Object*, ya que es el ancestro de cualquier clase en Java. Pero poder "parametrizar" de manera más específica el tipo (y no tan genérica como *Object*) nos permite controlar mejor los posibles errores en tiempo de compilación.

Aquí tenéis un ejemplo de declaración de una clase con "genéricos" donde sus métodos también contienen "genéricos". Se trata de una clase muy simple con un único atributo. Pero ese atributo, en lugar de ser Integer, LocalDate o Dispositivo, permito que a la hora de declarar una variable de esa clase, se pueda especificar de qué tipo quiero que sea:

**public** **class** **Caja**<T> {

**private** T atributo;

**public** **Caja** (T valorAtributo) {

**this**.atributo= valorAtributo;

}

**public** **void** **setAtributo** (T valorAtributo) {

**this**.atributo= valorAtributo;

}

**public** T **getAtributo** () {

**return** **this**.atributo;

}

}

Ahí podéis observar que no se indica de qué tipo es el atributo "atributo" de la clase Caja, sino que permito que pudiera ser cualquiera. Solo a la hora de declarar una variable de tipo Caja es cuando tendré que indicar de qué tipo quiero que sea el "genérico" <T>. Por ejemplo:

Caja<Integer> caja1;

Caja<String> caja2;

Caja<LocalDate> caja3;

caja1= **new** Caja<>(**25**);

caja2= **new** Caja<>("Caja2");

caja3= **new** Caja<>(LocalDate.of(**2022**,**01**,**01**));

System.out.printf ("Contenido de caja1: %d\n", caja1.getAtributo().intValue());

System.out.printf ("Contenido de caja2: %s\n", caja2.getAtributo());

System.out.printf ("Contenido de caja2: %s\n", caja3.getAtributo());

En este ejemplo de uso podéis observar cómo puedo declarar distintas variables de tipo Caja donde el atributo "genérico" o "parametrizable" puede ser del tipo que yo quiera. Eso sí, debe ser un objeto, no puede ser un tipo primitivo. Es decir, que si queremos que sea un *int, double, char, boolean*, etc. tendremos que echar mano de su clase "envoltorio" homóloga *(Integer, Double, Char, Boolean*, etc.) .

En una clase (o método) podemos tener tantos elementos (atributos para las clases o parámetros para los métodos) "genéricos" o "parametrizables" como deseemos. Por ejemplo, podríamos haber declarado una clase con dos atributos parametrizables (y algunos otros que no lo fueran):

**public** **class** **CajaDoble**<T1, T2> {

**private** T1 atributo1;

**private** T2 atributo2;

**private** **int** atributo3;

**public** **CajaDoble** (T1 atributo1, T2 atributo2, **int** atributo3) {

**this**.atributo1= atributo1;

**this**.atributo2= atributo2;

**this**.atributo3= atributo3;

}

**public** **void** **setAtributo1** (T1 valorAtributo) {

**this**.atributo1= valorAtributo;

}

**public** T1 **getAtributo1** () {

**return** **this**.atributo1;

}

}

En este último ejemplo podéis observar la implementación de una clase con dos elementos genéricos (o parametrizables). Dos de esos elementos serán los tipos de dos atributos, así como tipos en algunos parámetros de algunos métodos. Esto es lo fundamental que se cuenta en estos apartados.

**1.1.- Clases y métodos genéricos (II)**

¿Crees que 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 - 1 - i];

array[array.length - 1 - i] = 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 ello, va intercambiando entre sí el primer elemento con el último (cuando i=0),  el segundo con el penúltimo (cuando i=1), etc., hasta llegar a la mitad del array, momento en que se habrán intercambiado todos y el array estará invertido.

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 ("<strong>></strong>"), 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>()). Así, el objeto u es una instancia de la clase genérica Util particularizada para objetos de tipo Integer. Por eso podemos invocar al método u.invertir(numeros), de forma que el parámetro formal T[] array (un array de objetos de tipo T) se sustituye por el parámetro actual numeros, que es un array de objetos de tipo Integer.

Ahora bien, a partir de Java 7 es posible utilizar el ***operador diamante*** ("<>") para simplificar la instanciación o creación de nuevos objetos a partir de clases genéricas, de manera que la instanciación anterior podría quedar como new Util<>(), sin necesidad de especificar el tipo "concreto" Integer en la llamada al constructor. El ejemplo anterior completo quedaría entonces como sigue:

Integer[] numeros={0,1,2,3,4,5,6,7,8,9};

Util<Integer> u= new Util<>(); // Sólo a partir de Java 7

u.invertir(numeros);

for (int i=0;i<numeros.length;i++){ <br /> System.out.println(numeros[i]);<br />}

Por tanto, a partir de ahora, **siempre que nos sea posible utilizaremos el *operador diamante*** ("**<>**") para mejorar la legibilidad del código, aunque la otra opción es siempre posible (y necesaria si por alguna razón se fuera a trabajar con versiones anteriores a Java 7).

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.

Todavía hay un montón de cosas más sobre los métodos y las clases genéricas que deberías saber. En el siguiente documento se muestran algunos usos interesantes de los genéricos:

[Profundizando en las clases y métodos genéricos](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/PROG08_CONT_R027_digintogenerics_OK.pdf) (pdf - 82,72 KB).

Final del formulario

**2.- Introducción a las colecciones**

**Una colección a nivel de software es un grupo de elementos almacenados de forma conjunta en una misma estructura**.

Eso son 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 hacer uso de estos algoritmos.

Las colecciones son en general elementos de programación que están disponibles en muchos lenguajes de programación. En algunos de ellos 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 unidad.

Las colecciones en Java parten de una serie de interfaces básicas. Cada interfaz define un [modelo](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/#tf86d9a38-19a1-eff5-2f99-47d174fbfc88) 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(): devuelve el número de elementos de la colección.
* Método boolean isEmpty(): devuelve true si la colección está vacía.
* Método boolean contains (Object element): retornará true 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(): vacíar la colección.

**3.- Conjuntos (I). ¿Qué son y cómo se declaran? HashSet**

Los conjuntos son un tipo de colección que **no admite duplicados**, derivados del concepto matemático de conjunto.

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](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/#tc22abb21-a14c-a7da-c642-05c181eaccc5), lo cual acelera enormemente el acceso a los objetos almacenados. 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](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/#t50bb754e-6796-cd16-ac04-468a5433d0d1) 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](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/#t2b5e1546-0c47-04a0-eef6-61ad4d59152d). 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.

En este módulo no entramos en los detalles de cómo se implementan internamente las listas, ni los árboles, ni los algoritmos de ordenación y búsqueda que se implementan para conseguir que el almacenamiento y las búsquedas sean más o menos eficientes, solo nos interesa tener unas nociones de las ventajas e inconvenientes que presentan unas estructuras frente a otras, para elegirlas, y una vez elegidas, poder usarlas. Veamos un ejemplo de uso básico de la estructura HashSet y después, 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>();

**Dado que HashSet es una implementación de la interfaz Set, podemos también crearlo de la siguiente forma:**

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

**En este segundo ejemplo simplemente se cambia el tipo usado para la variable, así la variable conjunto podrá apuntar a cualquier implementación de la interfaz Set (HashSet, TreeSet, etc.).  ¡¡La potencia de usar interfaces en acción!!**

Después podremos ir almacenando objetos dentro del conjunto usando el método add() definido por la interfaz Set, ya que extiende a Collection. 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("No se pudo añadir. El número "+n+" ya está en la lista.");

}

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

Final del formulario

**3.1.- Conjuntos (II). ¿Cómo acceder a sus elementos?**

Para obtener todos los elementos almacenados en un conjunto hay que usar**iteradores**, que permiten obtener los elementos del conjunto uno a uno de forma secuencial (es la única forma de acceder a todos los elementos de un conjunto).

Los iteradores se verán en mayor profundidad más adelante, así que de momento los utilizaremos de forma "transparente" mediante un bucle **"for-each**" o bucle "para cada". Este tipo de bucles ya los vimos en la unidad dedicada a las estructuras de control, aunque apenas los habíamos utilizado hasta ahora. En el siguiente código tienes un ejemplo de uso de 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, donde:

* tipo es el tipo del objeto sobre el que se ha creado la colección;
* variable es la variable donde se almacenará cada elemento de la colección para ser procesado en cada iteración;
* colección es la colección en sí.

Los bucles**for-each** se pueden usar para todas las colecciones.

## Ejercicio resuelto

Realiza un pequeño programa 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**).

Package ejemplo;

import java.util.HashSet;

import java.util.Scanner;

public class Ejemplo {

public static void main(String[] args) {

HashSet<Integer> conjunto;

conjunto = new HashSet<>();

String str;

Scanner teclado= new Scanner(System.in);

do {

System.out.printf("Introduce el número %dº: ", (conjunto.size() + 1));

try {

str=teclado.nextLine();

Integer numero = Integer.parseInt(str);

if (!conjunto.add(numero)) {

System.out.printf("El número %d ya está en la lista. Debes introducir otro.\n",numero);

}

} catch (NumberFormatException e) {

System.out.printf("Número erróneo. Debe ser un número entero válido.\n");

}

} while (conjunto.size() < 5);

// Calcular la suma

int sumaElementosConjunto =0;

for (Integer elemento : conjunto) {

sumaElementosConjunto += elemento;

}

System.out.printf("La suma es: %d. \n\n", sumaElementosConjunto);

}

}

Y un ejemplo de ejecución:

Introduce el número 1º: 34

Introduce el número 2º: -34

Introduce el número 3º: 3.4

Número erróneo. Debe ser un número entero válido.

Introduce el número 3º: error

Número erróneo. Debe ser un número entero válido.

Introduce el número 3º: 10

Introduce el número 4º: 10

El número 10 ya está en la lista. Debes introducir otro.

Introduce el número 4º: 34

El número 34 ya está en la lista. Debes introducir otro.

Introduce el número 4º: 5

Introduce el número 5º: 15

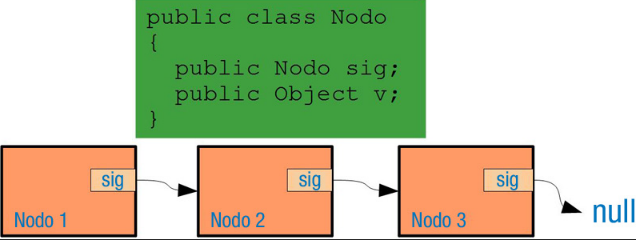
La suma es: 30.

Principio del formulario

Final del formulario

**3.2.- Conjuntos (III). ¿En qué se diferencian LinkedHashSet y TreeSet?**

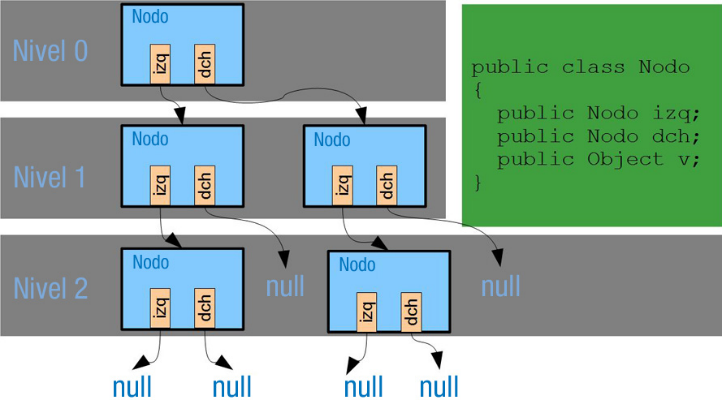
Ya se comentó antes: la diferencia se encuentra básicamente en **su funcionamiento interno**que las hace más o menos eficientes y por tanto más o menos adecuadas**según el tipo de operaciones más frecuentes** que necesitemos realizar con los elementos que contienen.

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 una referencia al siguiente nodo de la lista**. Si no hay siguiente nodo, se indica poniendo nulo (null) en la referencia al siguiente nodo.

Las listas enlazadas tienen un montón de operaciones asociadas en las que no vamos a profundizar: eliminación de un nodo de la lista, inserción de un nodo al final, al principio o entre dos nodos, etc.

**Gracias a las colecciones que nos proporciona Java podremos utilizar listas enlazadas sin tener que complicarnos, ni conocer, los detalles de su programación y funcionamiento interno, ya que nos las proporciona "listas para usar" cómodamente, con toda una serie de métodos disponibles, que nos proporciona su interfaz.**

La estructura TreeSet, 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, éste 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.

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**, como conjuntos que son, y se usan los mismos métodos ya vistos antes, los existentes en la interfaz Set (que es la interfaz que implementan).

|  | | |
| --- | --- | --- |
|  | **Conjunto TreeSet.** | **Conjunto LinkedHashSet.** |
| **Ejemplo de uso** | 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);  } | 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);  } |
| **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) |

En los ejemplos anteriores también se podría haber optado por usar una variable tipo Set. Por ejemplo, en el caso del TreeSet podría ser como sigue (con el mismo resultado):

Set <Integer> t;

t=new TreeSet<Integer>();

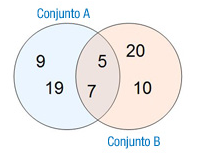
Final del formulario

**3.3.- Conjuntos (IV). Combinando datos de varias colecciones**

¿Cómo podría copiar los elementos de un conjunto de uno a otro?

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> conjuntoA= new TreeSet<Integer>();

conjuntoA.add(9); conjuntoA.add(19); conjuntoA.add(5); conjuntoA.add(7);

// Elementos del conjunto A: 9, 19, 5 y 7

LinkedHashSet<Integer> conjuntoB= new LinkedHashSet<Integer>();

conjuntoB.add(10); conjuntoB.add(20); conjuntoB.add(5); conjuntoB.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:

| **Combinación.** | **Código.** | **Elementos finales del conjunto A.** |
| --- | --- | --- |
| **Unión.**  **Añadir todos los elementos del conjunto B en el conjunto A.** | <code>conjuntoA.addAll(conjuntoB) | 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>conjuntoA.removeAll(conjuntoB) | Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19. |
| **Intersección.**  **Retiene los elementos comunes a ambos conjuntos.** | <code>conjuntoA.retainAll(conjuntoB) | 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**.

Final del formulario

**3.4.- Conjuntos (V). Ordenando sus elementos**

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 e implementar la interfaz Set, 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 objeto1, Objeto objeto2) { ... }

}

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

* Si el primer objeto (objeto1) se considera menor que el segundo (o<code>bjeto2), debe devolver un número entero negativo.
* Si el primer objeto (o<code>bjeto1) se considera mayor que el segundo (o<code>bjeto2), debe devolver un número entero positivo.
* Si ambos son iguales, debe devolver 0.

Te suena, ¿verdad? Es el mismo criterio que seguía el método compareTo() que usábamos para comparar dos String... pero eso es otra historia

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, pensando en menor y mayor,  puede ser un poco liosa, así que es recomendable en tales casos pensar en términos de delante y detrás, o de antes y después, más o menos de la siguiente forma:

* Si el primer objeto (o<code>bjeto1) debe ir antes que el segundo objeto (o<code>bjeto2), devolver entero negativo.
* Si el primer objeto (o<code>bjeto1) debe ir después que el segundo objeto (o<code>bjeto2), devolver entero positivo.
* Si ambos son iguales, debe devolver 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:

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

**Ejercicio resuelto**

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, éstos se tienen que ordenar de forma que la suma de sus atributos (a y **b**) sea descendente. ¿Cómo sería el comparador?

class ComparadorDeObjetos implements Comparator<Objeto> {

@Override

public int compare(Objeto objeto1, Objeto objeto2){

int resultado;

int sumaObjeto1=objeto1.a+objeto1.b;

int sumaObjeto2=objeto2.a+objeto2.b;

if (sumaObjeto1 < sumaObjeto2){

resultado=1;

}else{

if (sumaObjeto1>sumaObjeto2){

resultado=-1;

}else{

resultado=0;

}

}

return resultado;

}

}

Principio del formulario

Final del formulario

**4.- Listas (I). Definición y diferencias con los conjuntos**

La **interfaz**[***Collection***](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) es el padre de todas las colecciones e indica las acciones (métodos) que debe ser capaz de realizar cualquier colección. Una colección representa a un grupo de elementos.

Las hijas más conocidas (subinterfaces) de ***Collections*** son: (recordad que siguen siendo **interfaces**, no clases)

* [***List***](https://docs.oracle.com/javase/8/docs/api/java/util/List.html), para representar **listas**. Incluirán comportamientos (métodos) específicos de una lista (una secuencia de elementos).
* [***Set***](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html), para representar **conjuntos**. Incluirán comportamientos (métodos) específicos de un conjunto (grupo de elementos que no están en secuencia y no pueden repetirse: un elemento está o no está en un conjunto).
* [***Queue***](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html), para representar **colas**. Incluirán comportamientos (métodos) específicos de una cola (secuencia de elementos donde el último que llega se coloca siempre al final).
* ***Stack****,*para representar **pilas**.
* *etc.*

Cada una de esas subinterfaces será implementada por clases que implementan esos comportamientos manera u otra, dependiendo de cómo se represente "internamente" la estructura de datos y cómo se lleven a cabo unos métodos u otros:

* Para ***List***, tendremos las clases ***[ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html" \t "_blank" \o "class in java.util)***, ***[LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html" \t "_blank" \o "class in java.util)***, ***Stack***, ***Vector***, etc.
* Para ***Set***, tendremos las clases ***[HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html" \o "class in java.util" \t "_blank)***, ***[TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html" \t "_blank" \o "class in java.util)***, ***[LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html" \o "class in java.util" \t "_blank)***, etc.
* Para ***Queue***, tendremos las clases ***[LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html" \t "_blank" \o "class in java.util)***, ***PriorityQueue***, etc.
* etc.

**¿Qué es una lista?** Esencialmente se trata una secuencia de elementos. Algunos de los métodos más habituales que puede tener una lista (interfaz ***List***): ***add***, ***get***, ***remove***, ***set***, ***size***, ***indexOf***, ***isEmpty***, ***clear***, ***contains***, etc. Podemos observar que los arrays no son más que una primera forma "primitiva" de implementar listas.

¿Qué diferencia hay entre usar ***ArrayList***, ***LinkedList*** o alguna otra clase que implemente la interfaz ***List***? Pues la forma en la que están implementadas "por dentro". Para un programador que las use, desde fuera son exactamente iguales (igual interfaz: nombres de los métodos, parámetros y tipos de los métodos, tipos devueltos, atributos públicos, etc.). Por dentro se implementarán de una manera o de otra (usando arrays, usando enlaces de unos elementos a otros, etc.). Esto hará que en función de lo que queramos hacer, sea más apropiado elegir una u otra opción (unas ocuparán más espacio en memoria, otras ocuparán menos espacio pero quizá sean más lentas, otras serán más eficientes en algunos métodos, otras serán más eficientes en otros métodos, etc.).

¿Qué diferencia entre el concepto de **lista** (interfaz ***List*** en Java) y el concepto de **conjunto** (interfaz ***Set*** en Java)? Principalmente que los elementos son únicos y no se repiten. No hay una secuencia de elementos, no tienen posición. **Un elemento simplemente está o no está en el conjunto**.

Principal dificultad de los conjuntos: obtener los elementos de un conjunto. No tienen posiciones. Por tanto, necesitamos una forma alternativa de recorrerlos que no sea con un índice numérico. Tenemos al menos dos opciones:

* Bucle ***foreach***. (como hacíamos con los arrays)
* Uso de **iteradores**.

Introducción al concepto de **iterador** (interfaz ***[Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html" \o "interface in java.util" \t "_blank)***):

* Se trata de un mecanismo que nos permite **recorrer uno tras otro los distintos elementos que hay almacenados en una estructura de datos**. Esto puede ser especialmente útil en aquellos casos en los que no existan índices o esos índices no sean necesariamente correlativos (no sean números por ejemplo).
* Para obtener un iterador que nos permita recorrer los distintos elementos de una colección, disponemos del método (***iterator()***). Cualquier clase que implemente la interfaz *Collection* dispondrá de este método. Utilizando ese iterador se podrá ir recorriendo cada uno de los elementos de la estructura.
* Los métodos que vamos a utilizar de un iterador serán esencialmente: ***hasNext()*** y ***next()***. El primero se utiliza para saber si queda algún elemento por recorrer en la estructura (devuelve un ***boolean***). El segundo para obtener el siguiente elemento disponible en la estructura. Cada vez que hagamos un ***next()*** el "puntero" del iterador avanzará una posición. Si ya no quedan elementos el método ***hasNext()*** devolverá false y ***next()*** hará que salte una excepción *NoSuchElementException*. También tendrás el método ***remove()***.

Para ver esto de una manera más "visual", aquí tenéis el [**enlace una grabación de una sesión online de introducción a *List*y *Set***](https://eu-lti.bbcollab.com/recording/353f8d13ccec419b864c632dfd532250)que ya os pasé el otro día y que os puede venir bien.

Y aquí tenéis algunos vídeos más para reforzarlo:

* [Java. Programación genérica ArrayList I. Vídeo 161](https://www.cursosenhd.com/tutorial/programacion-generica-arraylist-i/).
* [Java. Programación genérica ArrayList II. Vídeo 162.](https://www.cursosenhd.com/tutorial/programacion-generica-arraylist-ii/)-> Cuidado: dice que los elementos de *ArrayList* no tienen índices asociados, y en realidad sí los tiene. De hecho tenemos el método *get(posicion)* y el método *add (posicion, valor)*.
* [Java. Programación genérica. ArrayList III. Iteradores. Vídeo 163](http://www.pildorasinformaticas.es/unit/java-programacion-generica-arraylist-iii-iteradores-video-163/?id=4843).
* [Java. Programación genérica Qué es Por qué utilizarla](https://www.cursosenhd.com/tutorial/programacion-generica-utilizarla/).

Vídeos de introducción a las **colecciones** (muy bueno para repasar *List* y *Set*).

* [Java. Colecciones I. Vídeo 179](http://www.pildorasinformaticas.es/unit/java-colecciones-i-video-179/?id=4843).
* [Java. Colecciones II. Vídeo 180](http://www.pildorasinformaticas.es/unit/java-colecciones-ii-video-180/?id=4843).
* [Java. Colecciones III. Métodos equals y hashCode . Vídeo 181](https://www.cursosenhd.com/tutorial/colecciones-iii-metodos-equals-hashcode/). Vienen bien para repasar la implementación del método ***equals***.
* [Java. Colecciones IV. Métodos equals y hashCode II .Vídeo 182](https://www.cursosenhd.com/tutorial/colecciones-iv-metodos-equals-hashcode-ii/).
* [Java. Colecciones V Iteradores. Vídeo 183](https://www.cursosenhd.com/tutorial/colecciones-v-iteradores/).

Y por último, algunos vídeos para entender la diferencia entre distintas implementaciones de ***List***: ***ArrayList*** (listas implementadas internamente con arrays) y ***LinkedList*** (listas implementadas mediante elementos enlazados): (no es obligatorio, y no nos hace falta para este curso, pero es interesante para aprender más y en el futuro os será útil)

* [Java. Colecciones VI LinkedList I. Vídeo 184](https://www.cursosenhd.com/tutorial/colecciones-vi-linkedlist-i/).
* [Java. Colecciones VII LinkedList II. Vídeo 185](https://www.cursosenhd.com/tutorial/colecciones-vii-linkedlist-ii/).

**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 objeto). El método indexOf() permite conocer la posición (**índice**) de un elemento (**objeto**), si dicho elemento no está en la lista retornará**-1**.
* int lastIndexOf(Object objeto). El método lastIndexOf() nos permite obtener la última ocurrencia del objeto en la lista (dado que la lista sí 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 está en la posición de índice 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.

Final del formulario

**4.1.- Listas (II). ¿Cómo se usan?**

Pues para usar una lista haremos uso de sus implementaciones LinkedList y **ArrayList**. Veamos un ejemplo de su uso y después obtendrás respuesta a esta pregunta.

Supongo que intuirás cómo se usan, pero nunca viene mal un ejemplo sencillo, que nos aclare las ideas. El siguiente ejemplo muestra cómo usar una LinkedList, pero valdría también para ArrayList (no olvides importar las clases java.util.LinkedList y **java.util.ArrayList** según sea necesario). En este ejemplo se usan los métodos de acceso posicional a la lista:

LinkedList<Integer> t=new LinkedList<Integer>(); // Declaración y creación del LinkedList de enteros.

t.add(1); // Añade un elemento al final de la lista.

t.add(3); // Añade otro elemento al final de la lista.

t.add(1,2); // Añade en la posición 1 el elemento 2.

t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.

t.remove(0); // Elimina el primer elemento de la lista.

int contador= 0;

for (Integer i: t) {

contador++;

System.out.println("Elemento " + contador + ": " + i); // Muestra cada elemento de la lista.

}

En el ejemplo anterior se realizan muchas operaciones,  ¿ pero cuál será el contenido de la lista al final?

Pues será 2, 3 y 5. En el ejemplo cabe destacar el uso del bucle **for-each**. Recuerda que se puede usar en cualquier colección.

Veamos otro ejemplo, esta vez con ArrayList, de cómo obtener la posición de un elemento en la lista:

ArrayList<Integer> miLista=new ArrayList<Integer>(); // Declaración y creación del ArrayList de enteros.

miLista.add(10);

miLista.add(11); // Añadimos dos elementos a la lista.

miLista.set(miLista.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazamos.

En el ejemplo anterior se emplea tanto el método indexOf() para obtener la posición de un elemento como el método set() para reemplazar el valor en una posición, una combinación muy habitual. Se generará un ArrayList que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

miLista.addAll(0, t.subList(1, t.size()));

Este ejemplo es especial porque usa sublistas. Se usa el método size() para obtener el tamaño de la lista. Después el método subList() para extraer una sublista de la lista (que incluía en origen los números 2, 3 y 5), desde la posición 1 hasta el final de la lista (lo cual dejaría fuera al primer elemento). Y por último, se usa el método addAll() para añadir todos los elementos de la sublista al ArrayList anterior.

Debes saber que **las operaciones aplicadas a una sublista repercuten sobre la lista original**. Por ejemplo, si ejecutamos el método clear() sobre una sublista, se borrarán todos los elementos de la sublista, pero también se borrarán dichos elementos de la lista original:

miLista.subList(0, 2).clear();

Lo mismo ocurre al añadir un elemento, se añade en la sublista y en la lista original.

**Ten en cuenta que, al igual que pasaba con los conjuntos, las variables y atributos que contendrán una lista podrán crearse también de la siguiente forma:**

List<Integer> miLista=new ArrayList<Integer>();

**En vez de usar como tipo de variable ArrayList, LinkedList, etc. puedes usar como tipo List, dado que todas las anteriores serán implementaciones de dicha interfaz.**

**Para saber más**

Las listas enlazadas son un elemento muy recurrido y su funcionamiento interno es complejo. Te recomendamos el siguiente artículo de la wikipedia para profundizar un poco más en las listas enlazadas y los diferentes tipos que hay.

[Listas enlazadas.](http://es.wikipedia.org/wiki/Lista_enlazada)

Final del formulario

**4.2.- Listas (III). Diferencias entre LinkedList y ArrayList**

Las LinkedList utilizan listas doblemente enlazadas, que son listas enlazadas (como se vio en un apartado anterior), pero que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos. Los nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento. Tener un doble enlace significa que en cada nodo se almacena la información de cuál es el siguiente nodo y además, de cuál es el nodo anterior. Si un nodo no tiene nodo siguiente o nodo anterior, se almacena null o nulo para ambos casos.

No es el caso de los ArrayList. Éstos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente para nosotros, no nos enteramos cuando se produce, pero eso redunda en una diferencia de rendimiento notable dependiendo del uso:

* Los ArrayList son más rápidos en cuanto a acceso a los elementos, acceder a un elemento según su posición es más rápido en un array que en una lista doblemente enlazada (hay que recorrer la lista).
* En cambio, eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada de cualquier tipo.

¿Y esto que quiere decir?

**Si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, conviene usar una lista enlazada (LinkedList), pero si no se van a realizar muchas eliminaciones, sino que solamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (ArrayList)**.

LinkedList tiene otras ventajas que pueden hacer aconsejable su uso. Implementa las interfaces java.util.Queue y **java.util.Deque**. Dichas interfaces permiten hacer uso de las listas como si fueran una cola de prioridad o una pila, respectivamente.

**Las colas**, también conocidas como colas de prioridad, son una lista, pero que aportan métodos para trabajar de forma diferente. ¿Tú sabes lo que es hacer cola para que te atiendan en una ventanilla? Pues igual: se trata de que el primero que llega es el primero en ser atendido (FIFO, o Primero en Entrar, Primero en Salir). Simplemente se aportan tres métodos nuevos: meter en el final de la lista (add() y **offer()**), sacar y eliminar el elemento más antiguo (poll()), y examinar el elemento al principio de la lista sin eliminarlo (peek()). Dichos métodos están disponibles en las listas enlazadas LinkedList:

* boolean add(E e) y boolean offer(E e), devolverán true si se ha podido insertar el elemento al final de la LinkedList.
* E poll() devolverá el primer elemento de la LinkedList y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
* E peek() devolverá el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

**Las pilas**, son todo lo contrario a las colas. Una pila es igual que una montaña de hojas en blanco (o una pila de platos): para añadir hojas nuevas (o platos),  se ponen encima del resto, y para retirar una se coge la primera que hay, encima de todas. En las pilas el último en llegar es el primero en ser atendido (LIFO). Para ello se proveen de tres métodos: meter al principio de la pila (push()), sacar y eliminar del principio de la pila (pop()), y examinar el primer elemento de la pila (peek(), igual que si usara la lista como una cola).

Ten en mente que tanto las colas como las pilas, son una lista enlazada sobre la que se hacen operaciones especiales.

Final del formulario

**4.3.- Listas (IV). ¿Es igual si los elementos son mutables o inmutables?**

A la hora de usar las listas, hay que tener en cuenta un par de detalles.

No es lo mismo usar las colecciones (listas y conjuntos) con objetos inmutables (String, **Integer**, etc.) que con objetos mutables. Los objetos inmutables no pueden ser modificados después de su creación, por lo que cuando se incorporan a la lista, a través de los métodos add(), se pasan por copia (es decir, se realiza una copia de los mismos). En cambio los objetos mutables (como las clases que tú puedes crear), **no se copian**, y eso puede producir efectos no deseados.

Imagínate la siguiente clase, que contiene un número:

class Test {

public Integer numero;

Test (int numero) {

this.numero=new Integer(numero);

}

}

La clase de antes es mutable, por lo que no se pasa por copia a la lista. Ahora imagina el siguiente código en el que se crea una lista que usa este tipo de objeto, y en el que se insertan dos objetos:

Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.

Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.

LinkedList<Test> lista=new LinkedList<>(); // Creamos una lista enlazada para objetos tipo Test.

lista.add(p1); // Añadimos el primer objeto test.

lista.add(p2); // Añadimos el segundo objeto test.

for (Test p:lista){

System.out.println(p.numero); // Mostramos la lista de objetos.

}

¿Qué mostraría por pantalla el código anterior?   Simplemente mostraría los números 11 y 12 en líneas consecutivas.

Ahora bien, ¿qué pasa si modificamos el valor de uno de los números de los objetos Test? ¿Qué se mostrará al ejecutar el siguiente código?

p1.numero=44;

for (Test p:lista){

System.out.println(p.numero);

}

El resultado de ejecutar el código anterior es que se muestran los números 44 y 12. El número ha sido modificado y no hemos tenido que volver a insertar el elemento en la lista para que en la lista se cambie también. Esto es porque en la lista no se almacena una copia del objeto Test, sino un apuntador o referencia a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).

Final del formulario

**5.- Conjuntos de pares clave/valor**

Otra estructura dinámica importante, que en Java no es hija de *Collection* (es decir, que estrictamente no es una "colección", aunque muchas veces hay quien la mete en el grupo de "colecciones"), son las ***listas de pares clave-valor (o key-value)***. Otros nombres con los que se les suelen llamar son ***mapas***, ***maps***, ***arrays asociativos***, ***tablas hash***, etc. La idea fundamental de estas estructuras es que cada elemento que se almacena (**valor** o **value**) está asociado con un valor único (**clave** o **key**). De este modo, para almacenar un valor en la estructura, tendremos que hacerlo indicando la clave, y lo mismo para recuperarlo. En Java estas estructuras están representadas por la **interfaz *[Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html" \o "interface in java.util" \t "_blank)***. Esa es la interfaz que define cuáles deben ser los comportamientos (métodos) de una clase que sea de tipo lista de pares clave-valor. En Java hay varias clases que implementan esta interfaz y que podemos usar. Entre ellas están: [***HashMap***](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), ***[TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html" \o "class in java.util" \t "_blank)***, ***[Hashtable](https://docs.oracle.com/javase/8/docs/api/java/util/Hashtable.html" \o "class in java.util" \t "_blank)***, ***[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html" \o "class in java.util" \t "_blank)***, etc.

Aquí tenéis el [enlace a una grabación de una sesión online](https://drive.google.com/file/d/1l-MnemEdtu0Smbxo2LioiPPaZxgf3Dcx/view?usp=sharing) de un año anterior donde estuvimos hablando sobre esta estructura.

Y aquí tenéis un vídeo de la serie "*Curso Completo de Java desde Cero*" donde también se habla del tema: [Java. Colecciones XI Mapas. Vídeo 189](https://www.cursosenhd.com/tutorial/colecciones-xi-mapas/).

Tenemos por un lado cada palabra y por otro su significado. Para resolver este problema existe precisamente un tipo de [array asociativo](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/#tbc4d4d16-3cec-c930-1d09-26ee976f35d6): los **mapas o diccionarios**, que permiten almacenar pares de valores conocidos como  **pares clave-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
* java.util.LinkedHashMap.

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 (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces devolverá el valor asociado que tenía antes, si la llave no existía, entonces devolverá null. |
| **V get(Object key)** | Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, devolverá null. |
| **V remove(Object key)** | Elimina la llave y el valor asociado. Devuelve el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe. |
| **boolean containsKey(Object key)** | Devolverá true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso. |
| **boolean containsValue(Object value)** | Devolverá true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso. |
| **int size()** | Devolverá el número de pares llave y valor almacenado en el mapa. |
| **boolean isEmpty()** | Devolverá true si el mapa está vacío, false en cualquier otro caso. |
| **void clear()** | Vacía el mapa. |

**Al igual que pasa con las colecciones anteriores (listas y conjuntos), puedes crear un mapa de la siguiente forma:**

**Map<String,Integer> miMapa=new HashMap<String,Integer>();**

**En vez de usar HashMap como tipo para la variable (HashMap miMapa por ejemplo), puedes usar Map (por ejemplo: Map miMapa). Esto puedes hacerlo porque  HashMap es una implementación de la interfaz Map, y lo mismo es aplicable para el resto de implementaciones de mapa: LinkedHashMap, TreeMap, etc.**

Final del formulario

**6.- Iteradores (I). ¿Qué son y cómo se usan?**

Son un mecanismo que nos permite recorrer todos los elementos de una colección de forma secuencial, sencilla y 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();

Fíjate 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. Si no 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(). Devolverá true si le quedan más elementos a la colección por visitar. false en caso contrario.
* E next(). Devolverá 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 incómoda 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 numero=it.next(); // Escogemos el siguiente elemento.

if (numero%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.

Las listas permiten acceso posicional a través de los métodos get() y **set()**, y acceso secuencial a través de iteradores.

¿Cuál es para ti 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())"?

**6.1.- Iteradores (II). ¿Qué características debemos considerar?**

¿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. Se eliminan de la lista aquellos que son pares y sólo 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  que al no especificar el tipo de objeto del iterador,  tenemos que usar la conversión explícita de tipos (**casting**) en la línea 7.

|  |  |
| --- | --- |
| **Ejemplo indicando el tipo de objeto de iterador** | **Ejemplo no indicando el tipo de objeto del iterador** |
| ArrayList <Integer> lista=new ArrayList<>();  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();  } | ArrayList <Integer> lista=new ArrayList<>();  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();  } |

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>();

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 debes 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.

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.

Principio del formulario

Final del formulario

Final del formulario

**7.- Algoritmos (I). ¿Qué podemos hacer con las colecciones?**

En el primer subapartado se os introduce a la clase ***Collections*** (acabado en -s), así como de la clase *Arrays*, que ya vimos en la unidad 4. Se trata de clases de tipo "caja de herramientas" que no son instanciables y que contienen una serie de utilidades para ser usadas por las colecciones y otras estructuras de datos. Esas utilidades se encuentran implementadas en forma de **métodos estáticos**. Otra clase de tipo "caja de herramientas" que llevamos utilizando prácticamente desde el comienzo de curso es la clase *Math*, que nos proporciona todo un repertorio funciones matemáticas.

En el siguiente subapartado (7.1) se habla de las interfaces ***Comparable<T>*** y ***Comparator<T>***. Estas dos interfaces son, junto con los algoritmos proporcionados por *Collections*, las protagonistas de toda la sección. Consisten en dos formas diferentes de proporcionar un mecanismo de "comparación" entre objetos de una clase para que luego pueda llevarse a cabo una ordenación. Si dos elementos no disponen de un criterio de comparación, no podrían ordenarse. Lo que permiten estas interfaces es implementar ese criterio (en un caso mediante el método ***compareTo*** y en el otro mediante el método ***compare***).

Es muy importante que sigáis el ejemplo que se propone con la clase *Articulo* del **apartado 7.1** ("***Algoritmos (II). Ordenando los elementos de la colección***"). Una vez lo hayáis leído, entonces podéis echarle un vistazo a la grabación de una sesión online de un año anterior donde estuvimos hablando sobre las interfaces *Comparable* y *Comparator* y cómo implementar comparadores:

* [**Ordenación de una lista. Interfaces *Comparable* y *Comparator***](https://drive.google.com/file/d/1kCm5HmGmd0DdvIj2YC1pl8gJPQuD8_a8/view?usp=sharing).

En el último subapartado de la sección (7.2) es una visión general de las herramientas que proporcionan las clases "toolbox" o "caja de herramientas" *Collections* y *Arrays* (recuerda, ambas en plural, terminadas en -s).

**Ejercicio Resuelto**

(ver ejercicio resuelto apartado 7 de la unidad)

Principio del formulario

Final del formulario

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 cómo ordenarlos. Como se explicó 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):

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

**7.1.- Algoritmos (II). Ordenando los elementos de la colección**

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan.

**Ada** pidió a **Ana** que los artículos del pedido aparecieran ordenados por código de 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 Articulo {

public String codigoArticulo; // 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, lo que supone implementar 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 articulo1, Articulo articulo2) {

return articulo1.codigoArticulo.compareTo(articulo2.codigoArticulo);

}

}

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 Articulo implements Comparable<Articulo>{

public String codigoArticulo;

public String descripcion;

public int cantidad;

@Override

public int compareTo(Articulo articulo) {

return codigoArticulo.compareTo(articulo.codigoArticulo);

}

}

Del ejemplo anterior se pueden resaltar 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 Articulo 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 el objeto que se pasa por parámetro es igual al objeto ha llamado al método, se tendría que devolver  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);"

Final del formulario

**7.2.- Algoritmos (III). Algunas operaciones adicionales con colecciones**

Una vez vista la ordenación, que quizás es 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. | Collections.shuffle (lista); |
| **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. | Collections.fill (lista,elemento);  Arrays.fill (array,elemento); |
| **Búsqueda binaria.** | Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito. | Collections.binarySearch(lista,elemento);  Arrays.binarySearch(array, elemento); |
| **Convertir un array a lista.** | Permite rápidamente convertir un array a una lista de elementos, y es extremadamente útil. No se especifica el tipo de lista retornado (no es ArrayList ni **LinkedList**), Solo se especifica que retorna una lista que implementa la interfaz java.util.List. | List lista=Arrays.asList(array);  Si el tipo de dato almacenado en el array es conocido (Integer por ejemplo), es conveniente especificar el tipo de objeto de la lista:  List<Integer>lista = Arrays.asList(array); |
| **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 Collections, sino propio de la interfaz Collection. Es conveniente que sepas de su existencia. | 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:  Integer[] array=new Integer[lista.size()];  lista.toArray(array); |
| **Dar la vuelta.** | Da la vuelta a una lista, poniéndola en orden inverso al que tiene. | Collections.reverse(lista); |

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 requiere conocer el funcionamiento de los arrays y de las expresiones regulares para su uso. 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!

En el siguiente vídeo podrás ver en qué consiste la búsqueda binaria y cómo se aplica de forma sencilla:

[Resumen textual alternativo para "Método de búsqueda binaria"](https://educacionadistancia.juntadeandalucia.es/formacionprofesional/pluginfile.php/55853/mod_scorm/content/0/PROG09_Descripcion_Video_metodo_de_busqueda_binaria.html)

Ahora bien, este vídeo contiene un par de erratas. A ver si eres capaz de encontrarlas. Te damos una pista: la primera se encuentra sobre el minuto y medio, y la segunda sobre los dos minutos y medio.