

## Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

**Ana** sigue con la pequeña aplicación de procesamiento de pedidos que le pidió se jefa **María**.

En este caso, después de comprobar que el pedido tiene el formato correcto, algo que pensó hacer con expresiones regulares, llega el momento de almacenar en alguna estructura en memoria principal, si es posible de forma ordenada.

Ana ya conoce los arrays, como estructura de almacenamiento de datos en memoria pero ha llegado a la conclusión de que tienen ciertas limitaciones, sobre todo la imposibilidad de crecer en tiempo de ejecución y de mantener la estructura ordenada. Por otro lado, es bastante compleja la eliminación de elementos.

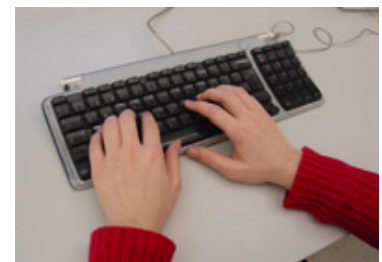
- Había pensado en utilizar arrays, es la única estructura de datos que conozco, pero hay limitaciones que no sé como solventar - comentó **Ana**. ¿Existirá alguna estructura de datos que no presente este tipo de limitaciones?-

- Si Ana, al igual que en otros lenguajes de programación, en Java se pueden utilizar estructuras de datos dinámicas. Pueden crecer en tiempo de ejecución tanto que posibilidades tenga la memoria y además Java proporciona una serie de API para su manejo con mucha funcionalidad - le sugirió **María**.

- ¡Pues estoy deseando conocer esa API! -dijo Ana.

Vamos a ello, te sorprenderás de la cantidad de funcionalidad que está a disposición de los programadores.

Como ya estudiamos en una unidad anterior, cuando el volumen de datos a manejar por una aplicación es elevado, no basta con utilizar variables. Manejar los datos de un único pedido en una aplicación puede ser relativamente sencillo, pues un pedido está compuesto por una serie de datos y eso simplemente se traduce en varias variables. Pero, ¿qué ocurre cuando en una aplicación tenemos que gestionar varios pedidos a la vez?. ¿De qué herramientas disponemos cuando necesitamos almacenar muchos datos, tanto simples como compuestos?. La solución propuesta hasta el momento pasa por utilizar estructuras de datos definidas por el usuario, tal y como vimos en la unidad 6. Además, el programador debe desarrollar también los algoritmos que procesen esas estructuras de datos.



ITE. Óscar Javier Estupiñán Estupiñán.  
id=133827 ([CC BY-NC](#))

La mayoría de lenguajes de programación proporcionan dentro de su API conjuntos de clases e interfaces que liberan al programador de la necesidad de definir diferentes tipos de datos compuestos, como pueden ser listas, conjuntos, etc. En el caso de Java tenemos el **Framework Collection**. Lo estudiaremos en profundidad a lo largo de esta unidad.

## Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](#)

# 1.- Introducción a las colecciones.

## Caso práctico

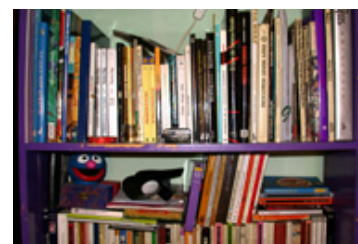
A **Ana** las listas siempre se le han atragantado, por eso no las usa. Después de darle muchas vueltas, ha pensado que no le queda más remedio y que tendrá que usarlas para almacenar los artículos del pedido. Además, ha concluido que es la mejor forma de gestionar un grupo de objetos, aunque sean del mismo tipo.

No sabe si lo más adecuado es usar una lista u otro tipo de colección, así que ha decidido revisar todos los tipos de colecciones disponibles en Java, para ver cuál se adecua mejor a sus necesidades.



LatinStock (CC BY-NC)

¿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.



ITE idITE=109532 (CC BY-NC)

Una **colección o contenedor** es un objeto que agrupa elementos múltiples en un objeto simple. Las colecciones son usadas para almacenar, recuperar y manipular datos.

Aunque un array o vector podría ser una colección, no está incluido en el framework Collections.

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 hecho 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):

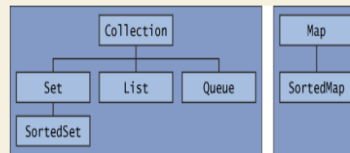
- ✔ Método `int size()`: retorna el número de elementos de la colección.
- ✔ Método `boolean isEmpty()`: retornará verdadero si la colección está vacía.
- ✔ Método `boolean contains (Object element)`: retornará verdadero si la colección tiene el elemento pasado como parámetro.
- ✔ Método `boolean add(E 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()`: vaciar la colección.

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

Los principales beneficios de utilizar este framework son:

1. **Reducen el esfuerzo de programación:** Disponen de un conjunto amplio de estructuras de datos y algoritmos, eficientes y probados.
2. **Incrementan la calidad de las aplicaciones.**
3. **Incrementan la interoperabilidad,** pues muchas APIs utilizan clases o interfaces de Collections como argumentos en sus métodos.
4. **Reducen el esfuerzo de aprender nuevas APIs.**
5. **Contribuyen a la reusabilidad del código.**

~ **Core Collection Interfaces:** Contiene diferentes tipos de colecciones y suponen el fundamento del *Framework Collection*.



Se trata de Interfaces que forman una jerarquía

Todas las interfaces son genéricas → pueden almacenar cualquier tipo de datos.

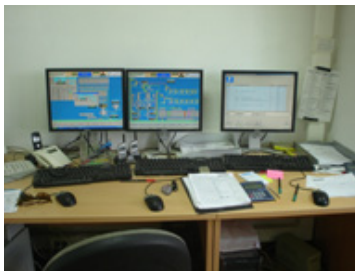
```
public interface Collection<E>...
```

Parámetro genérico → especificaremos al instanciar la colección el tipo de datos contenido.

Ministerio de Educación y FP ([CC BY-NC](#))

## 2.- Clases y métodos genéricos (I).

### Caso práctico



ITE. Esther Balgoma Hernando. idITE=179681  
([CC BY-NC](#))

**María** se acerca a la mesa de **Ana**, quiere saber cómo lo lleva:

-¿Qué tal? ¿Cómo vas con la tarea? -pregunta María.

-Bien, creo. Mi programita ya sabe procesar el archivo de pedido y he creado un par de clases para almacenar los datos, pero no sé cómo almacenar los artículos del pedido, porque son varios -comenta Ana.

-Pero, ¿cuál es el problema? Eso es algo sencillo.

-Pues que tengo que crear un array para guardar con los artículos del pedido, y no sé cómo averiguar el número de artículos antes de empezar a procesarlos. Es necesario saber el número de artículos para crear el array del tamaño adecuado.

-Pues en vez de utilizar un array, podrías utilizar una lista.

¿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 cómo transformar un método normal en genérico:



Salvador Romero Villegas ([CC BY-NC](#))



## Versión no Genérica

```
public class util {  
  
    public static int compararTamano(Object[] a, Object[] b) {  
        return a.length-b.length;  
    }  
}
```

## Versión Genérica

```
public class util {  
  
    public static <T> int compararTamano (T[] a, T[] b) {  
        return a.length-b.length;  
    }  
}
```

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 ( $\tau$ ) 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 que y mayor que ("<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.
<pre>Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.compararTamano ((Object[])a, (Object[])b);</pre>	<pre>Integer []a={0,1,2,3,4}; Integer []b={0,1,2,3,4,5}; util.&lt;Integer&gt;compararTamano (a, b);</pre>

## 2.1.- 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:



[Berteun](#). (Dominio público)

```
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.

# Autoevaluación

Dada la siguiente clase, donde el código del método `prueba` carece de importancia, ¿podrías decir cuál de las siguientes invocaciones es la correcta?

```
public class Util {  
  
    public static <T> int prueba (T t) { ... }  
  
};
```

- ☐ `Util.<int>prueba(4);`
- ☐ `Util.<Integer>prueba(new Integer(4));`
- ☐ `Util u=new Util(); u.<int>prueba(4);`

Incorrecto. No se pueden usar tipos de datos primitivos en los genéricos.

Correcto. Has captado la idea.

Incorrecto. Fíjate que `prueba` es un método estático y no se puede invocar así. Además se usan datos primitivos en un genérico, cosa que no es posible.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

## 3.- Conjuntos (I).

### Caso práctico

**Ana** se toma un descanso, se levanta y en el pasillo se encuentra con **Juan**, con el que entabla una conversación bastante amena. Una cosa lleva a otra y al final, Ana saca el tema que más le preocupa:



—¿Cuántos tipos de colecciones hay? ¿Tu te los sabes?  
—pregunta Ana.

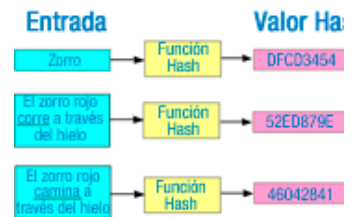
—¿Yo? ¡Que va! Normalmente consulto la documentación cuando los voy a usar, como todo el mundo. Lo que sí creo recordar es que había cuatro tipos básicos: los conjuntos, las listas, las colas y alguno más que no recuerdo. ¡Ah sí!, los mapas, aunque creo que no se consideraban un tipo de colección. ¿Por qué lo preguntas?

—Pues porque tengo que usar uno y no sé cuál.



¿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.

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:



[Fercufer \(CC BY-SA\)](#)

- ✓ `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 mas lenta que HashSet.
- ✓ `java.util.TreeSet`. Conjunto que almacena los objetos usando unas estructuras conocidas como árboles rojo-negro. Son más lentas que los dos tipos anteriores. pero tienen una gran ventaja: **los datos almacenados se ordenan por valor**. Es decir, que aunque se inserten los elementos de forma desordenada, internamente se ordenan dependiendo del valor de cada uno.

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

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

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

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

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

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

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

## Autoevaluación

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



- ☐ **HashSet.**
- ☐ **LinkedHashSet.**
- ☐ **TreeSet.**

No es correcto. Lee de nuevo los contenidos.

Incorrecto. Revisa el apartado y averigua por qué.

Efectivamente, has captado la idea a la primera.

## Solución

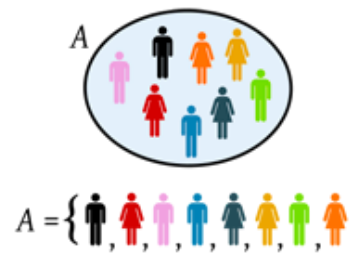
1. Incorrecto
2. Incorrecto
3. Opción correcta

### 3.1.- 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.



[Ilustración de persona: AIGA symbol signs collection; trabajo derivado por kismalac \(CC BY-SA\)](#)

## 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`).

#### Mostrar retroalimentación

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

```
public class EjemploHashSet {
    // ...
    // Ignorar args the command line arguments
    // ...
    public static void main(String[] args) {
        // Here some application logic here
        HashSet<Integer> conjuntoNumeros = new HashSet<>();
        String str;
        int i;
        while (i < 5) {
            JOptionPane.showMessageDialog("Introduce un número "+(conjuntoNumeros.size()+1)+"");
            Integer en = Integer.parseInt(str);
            if (conjuntoNumeros.add(en)) {
                JOptionPane.showMessageDialog("Número ya en la lista. Debes introducir otro.");
            }
            // ...
        }
        // ...
        // Calcular la suma
        Integer suma = 0;
        for (Integer i : conjuntoNumeros) {
            suma += i;
        }
        JOptionPane.showMessageDialog("La suma es: "+suma);
    }
}
```

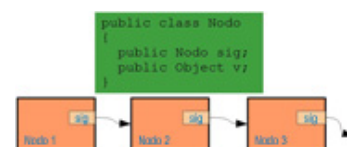
Salvador Romero Villegas y los autores de la herramienta de Software Libre NetBeans IDE 7.0 ([GNU/GPL](#))

#### Ejemplo con HashSet

## 3.2.- Conjuntos (III).

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

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.



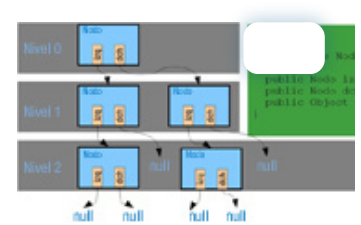
Salvador Romero Villegas ([CC BY-NC](#))

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 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).



Salvador Romero Villegas ([CC BY-NC](#))

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.

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**

```
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);
```

*Salida por pantalla*

1 3 4 99

(el resultado sale ordenado por valor)

#### Conjunto **LinkdetHashSet**

```
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);
```

*Salida por pantalla*



## Autoevaluación

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

- ☐ Verdadero.
- ☐ Falso.

Acertaste.

Incorrecto. Revisa el tema para entender en que has fallado.

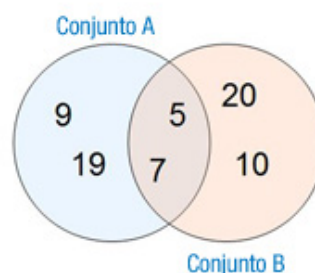
## Solución

1. Opción correcta
2. Incorrecto

## 3.3.- Conjuntos (IV).

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

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



```
TreeSet<Integer> A= new TreeSet<Integer>();
```

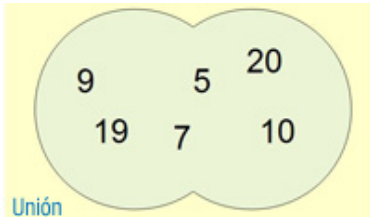
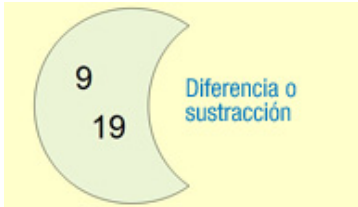

```
A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7
```

```
LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();
```

```
B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

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

## Tipos de combinaciones.

Combinación.	Código.	Elementos finales del conjunto A.
<b>Unión. Añadir todos los elementos del conjunto B en el conjunto A.</b>	<code>A.addAll(B)</code>	<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> <p>Ministerio de Educación y FP (<a href="#">CC BY-NC</a>)</p>
<b>Diferencia. Eliminar los elementos del conjunto B que puedan estar en el conjunto A.</b>	<code>A.removeAll(B)</code>	<p>Todos los elementos del conjunto A, que no estén en el conjunto B: 9, 19.</p>  <p>Diferencia o sustracción</p> <p>Ministerio de Educación y FP (<a href="#">CC BY-NC</a>)</p>
<b>Intersección. Retiene los elementos comunes a ambos conjuntos.</b>	<code>A.retainAll(B)</code>	<p>Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.</p>  <p>Intersección</p> <p>Ministerio de Educación y FP (<a href="#">CC BY-NC</a>)</p>



## Para saber más

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

[Álgebra de conjuntos.](#)

## Autoevaluación

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

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

No es correcto, no se trata de hacer una intersección.

Efectivamente, sería una diferencia.

Incorrecto. Como se ha dicho antes estas operaciones son comunes a todas las colecciones.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto



## 3.4.- 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;
```

en:Joestape89  
(CC BY-SA)

}

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?

#### Mostrar retroalimentación

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;  
  
    }  
  
}
```

## 4.- Listas (I).

### Caso práctico

**Juan** se queda pensando después de que Ana le preguntara si sabía los tipos de colecciones que había en Java. Obviamente no lo sabía, son muchos tipos, pero ya tenía una respuesta preparada:

—Bueno, sea lo que sea, siempre puedes utilizar una lista para almacenar lo que sea. Yo siempre las uso, pues te permiten almacenar cualquier tipo de objeto, extraer uno de las lista sin tener que recorrerla entera, buscar si hay o no un elemento en ella, de forma cómoda. Son para mi el mejor invento desde la rueda —dijo Juan.



Ministerio de Educación. [\(CC BY-NC\)](#)

—Ya, supongo, pero hay dos tipos de listas que me interesan, `LinkedList` y `ArrayList`, ¿cuál es mejor? ¿Cuál me conviene más? —respondió Ana.

¿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 si 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.

## Autoevaluación

Si `M` es una lista de números enteros, ¿sería correcto poner `"M.add(M.size(),3);"`?

- ☐ Sí.
- ☐ No.

Correcto. Inserta un elemento al final de la lista y es equivalente a poner `M.add(3)`.

Incorrecto. Piénsalo, ese código inserta un elemento al final de la lista.

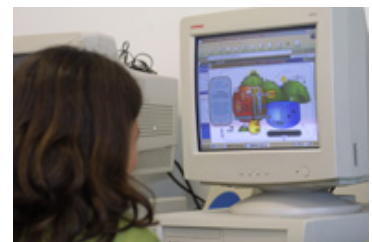
# Solución

1. Opción correcta
2. Incorrecto

## 4.1.- Listas (II).

Y, ¿cómo se usan las listas? 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 como se usan, pero nunca viene mal un ejemplo sencillo, que nos aclare las ideas. El siguiente ejemplo muestra como usar un `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:



ITE. Francisco Javier Martínez Adrados.  
idITE=148928 ([CC BY-NC](#))

```
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 elementos de la lista.

for (Integer i: t) System.out.println("Elemento:" + i); // Muestra la lista.
```

En el ejemplo anterior, se realizan muchas operaciones, ¿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> al=new ArrayList<Integer>(); // Declaración y creación del ArrayList de enteros.

al.add(10); al.add(11); // Añadimos dos elementos a la lista.

al.set(al.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. El ejemplo anterior generará un ArrayList que contendrá dos números, el 10 y el 12. Veamos ahora un ejemplo algo más difícil:

al.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 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:

```
al.subList(0, 2).clear();
```

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

## Debes conocer

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.](#)

## Autoevaluación

**Completa con el número que falta.**

**Dado el siguiente código:**

```
LinkedList<Integer> t=new LinkedList<Integer>();
```

```
t.add(t.size()+1); t.add(t.size()+1); Integer suma = t.get(0) + t.get(1);
```

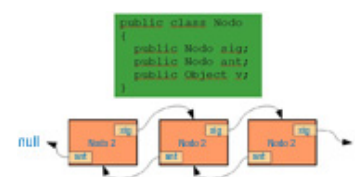
El valor de la variable `suma` después de ejecutarlo es .

Enviar

## 4.2.- Listas (III).

¿Y en qué se diferencia un `LinkedList` de un `ArrayList`? Los `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`. Estos se implementan utilizando arrays que se van redimensionando conforme se necesita más espacio o menos. La redimensión es transparente a nosotros, no nos enteramos cuando se produce, pero eso redundará en una diferencia de rendimiento notable



Salvador Romero Villegas (CC BY-NC)

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 (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? **Que 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 nos puede llevar a 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 que primero llega es el primero en ser atendido (**FIFO**). 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)`**, retornarán `true` si se ha podido insertar el elemento al final de la `LinkedList`.
- ✓ **`E poll()`** retornará 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()`** retornará el primer elemento de la `LinkedList` pero no lo eliminará, permite examinarlo. Retornará `null` si la lista está vacía.

Las pilas, mucho menos usadas, son todo lo contrario a las listas. Una pila es igual que una montaña de hojas en blanco, para añadir hojas nuevas 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. 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).

Las pilas se usan menos y haremos menos hincapié en ellas. Simplemente ten en mente que, tanto las colas como las pilas, son una lista enlazada sobre la que se hacen operaciones especiales.

## Autoevaluación

Dada la siguiente lista, usada como si fuera una cola de prioridad, ¿cuál es la letra que se mostraría por la pantalla tras su ejecución?

```
LinkedList<String> tt=new LinkedList<String>();
```

```
tt.offer("A"); tt.offer("B"); tt.offer("C");
```

```
System.out.println(tt.poll());
```

- ☐ A.
- ☐ C.
- ☐ D.

Correcto. Efectivamente, el primero en entrar es el primero que sale.

Incorrecto. Revisa el funcionamiento de las colas.



No es correcto. El elemento D no está en la lista.

## Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

### 4.3.- Listas (IV).

A la hora de usar las listas, hay que tener en cuenta un par de detalles, ¿sabes cuáles? Es sencillo, pero importante.

No es lo mismo usar las colecciones (listas y conjuntos) con objetos inmutables (**Strings**, **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.



ITE. Francisco Javier Martínez Adrados.  
idITE=148958 ([CC BY-NC](#))

Imaginate la siguiente clase, que contiene un número:

```
class Test
{
    public Integer num;

    Test (int num) { this.num=new Integer(num); }
}
```

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

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

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

for (Test p:lista) System.out.println(p.num); // Mostramos la lista de objetos.
```

¿Qué mostraría por pantalla el código anterior? Simplemente mostraría los números 11 y 12. 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.num=44;

```
for (Test p:lista) System.out.println(p.num);
```

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 a dicho objeto (solo hay una copia del objeto a la que se hace referencia desde distintos lugares).

## Citas para pensar

"Controlar la complejidad es la esencia de la programación."

*Brian Kerniga*

## Autoevaluación

Los elementos de un `ArrayList` de objetos `Short` se copian al insertarse al ser objetos mutables. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

Lo siento, pero no has acertado. Los objetos `Short` son inmutables.

Acertaste. Los elementos se pasan por copia por ser inmutables, no mutables.

### Solución

1. Incorrecto
2. Opción correcta

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

## Autoevaluación

Completa el siguiente código para que al final se muestre el número 40 por pantalla:

```
HashMap< String, [ ] > datos=new [ ] < String,String >();datos.  
[ ] ("A","44");System.out.println(Integer. [ ] (datos. [ ] (" [ ]  
"))-[ ] );
```

Enviar

## 6.- Iteradores (I).

### Caso práctico

Juan se acerco a la mesa de Ana y le dijo:

—María me ha contado la tarea que te ha encomendado y he pensado que quizás te convendría usar mapas en algunos casos. Por ejemplo, para almacenar los datos del pedido asociados con una etiqueta: nombre, dirección, fecha, etc.

—La verdad es que pensaba almacenar los datos del pedido en una clase especial llamada Pedido. No tengo ni idea de que son los mapas -dijo Ana-, supongo que son como las listas, ¿tienen iteradores?

—Según me ha contado María, no necesitas hacer tanto, no es necesario crear una clase específica para los pedidos. Y respondiendo a tu pregunta, los mapas no tienen iteradores, pero hay una solución... te explico.



Ministerio de Educación. ITE. idITE=175787  
(CC BY-NC)

¿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:



Java. (usuario wikipedia que subió la imagen: Machro) (Dominio público)

- ✓ **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())"**?

## 6.1.- 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 primer ejemplos se especifica el tipo de objeto del iterador, en el segundo ejemplo 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 del iterador

```
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();
```

```
}
```

### Ejemplo no indicando el tipo de objeto del iterador

```
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();
```

```
}
```

Un iterador es seguro porque esta 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 incomoda 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 como 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.

## Autoevaluación

¿Cuándo debemos invocar el método `remove()` de los iteradores?

- ☐ En cualquier momento.
- ☐ Después de invocar el método `next()`.
- ☐ Después de invocar el método `hasNext()`.
- ☐ No es conveniente usar este método para eliminar elementos, es mejor usar el de la colección.

Incorrecto. Deberías revisar el tema.

Efectivamente, has dado en el clavo.

No es correcto.

Incorrecto. Es relativo, si usas iteradores deberías usar este método, sino usas iteradores, no importa.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## Para saber más

A partir de Java 8, existen los conocidos **Java Stream Foreach**. Se trata de una nueva construcción que permite, entre otras cosas, recorrer colecciones utilizando una sintaxis muy cómoda y funcional.

En el siguiente enlace tienes una comparación entre los tradicionales bucles **Foreach** y **Java Stream Foreach**.

[Java Stream Foreach](#)

## 7.- Algoritmos (I).

### Caso práctico

**Ada** se acercó a preguntar a **Ana**. **Ada** era la jefa y **Ana** le tenía mucho respeto. **Ada** le preguntó cómo llevaba la tarea que le había encomendado **María**. Era una tarea importante, así que prestó mucha atención.

**Ana** le enseñó el código que estaba elaborando, le dijo que en un principio había pensado crear una clase llamada Pedido, para almacenar los datos del pedido, pero que Juan le recomendó usar mapas para almacenar los pares de valor y dato. Así que se decantó por usar mapas para ese caso. Le comentó también que para almacenar los artículos si había creado una pequeña clase llamada Artículo. Ada le dio el visto bueno:

—Pues Juan te ha recomendado de forma adecuada. Eso sí, sería recomendable que los artículos del pedido vayan ordenados por código de artículo —dijo Ada.

—¿Ordenar los artículos? Vaya, qué jaleo -respondió Ana.

—Arriba ese ánimo mujer, si has usado listas es muy fácil.

Ada explicó a Ana cómo mantener los artículos de un pedido ordenados por código de artículo. Inmediatamente después trató de dar implementación.



Ministerio de Educación (CC BY-NC)

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.



ITE. Ministerio de educación  
idITE=110292 (CC BY-NC)

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 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

```
Integer[] array={10,9,99,3,5};
```

```
Arrays.sort(array);
```

### Ejemplo de ordenación de una lista con números

```
ArrayList<Integer> lista=new ArrayList<Integer>();
```

```
lista.add(10); lista.add(9);lista.add(99);
```

```
lista.add(3); lista.add(5);
```

```
Collections.sort(lista);
```

## 7.1.- Algoritmos (II).

---

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. ¿Recuerdas la tarea que 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 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); }  
  
}
```



ITE. Ministerio de educación. idITE=112221  
(CC BY-NC)

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 codArticulo;

    public String descripcion;

    public int cantidad;

    @Override

    public int compareTo(Articulo o) { return codArticulo.compareTo(o.codArticulo); }

}
```

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 `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 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);`

## Autoevaluación

**Si tienes que ordenar los elementos de una lista de tres formas diferentes, ¿cuál de los métodos anteriores es más conveniente?**

- ☐ Usar comparadores, a través de la interfaz `java.util.Comparator`.
- ☐ Implementar la interfaz `comparable` en el objeto almacenado en la lista.

Efectivamente, creas tres comparadores, uno para cada forma de ordenar la lista.

No es la mejor forma, solo podrías ordenar los elementos de una forma.

## Solución

1. Opción correcta
2. Incorrecto

## 7.2.- 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
<b>Desordenar una lista.</b>	Desordena una lista, este método no está disponible para arrays.	<code>Collections.shuffle (lista);</code>
<b>Rellenar una lista o array.</b>	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>
<b>Búsqueda binaria.</b>	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.	<code>Collections.binarySearch(lista,elemento);</code> <code>Arrays.binarySearch(array, elemento);</code>
<b>Convertir un array a lista.</b>	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> .	<code>List lista=Arrays.asList(array);</code>  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: <code>List&lt;Integer&gt;lista = Arrays.asList(array);</code>
<b>Convertir una lista a array.</b>	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.	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: <code>Integer[] array=new Integer[lista.size()];</code>  <code>lista.toArray(array)</code>
<b>Dar la vuelta.</b>	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	<code>Collections.reverse(lista);</code>

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!



ITE. Ministerio de educación. idITE=109332  
(CC BY-NC)

## Para saber más

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

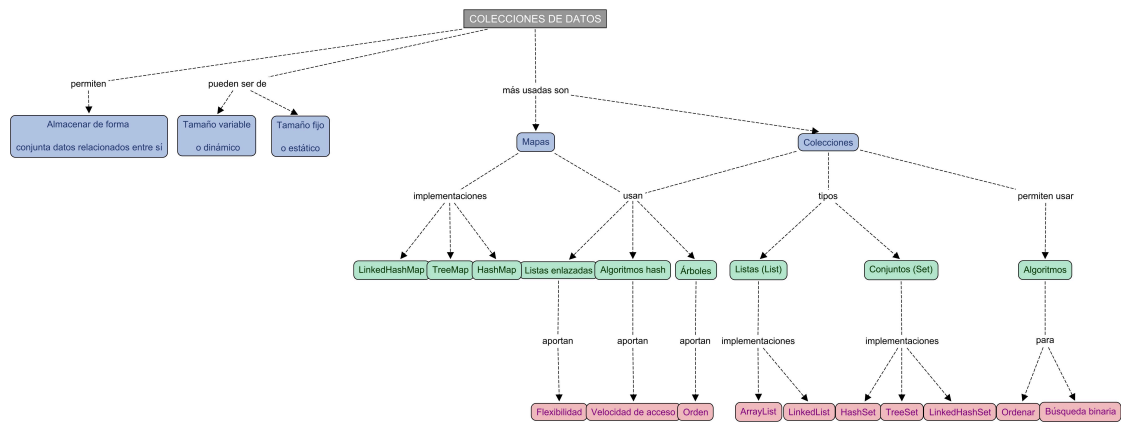
<https://www.youtube.com/embed/-isTI614INQ>

[Resumen textual alternativo](#)

## 8.- Conclusiones.

Durante el desarrollo de esta unidad hemos trabajado con una de APIs mas utilizadas de Java: el API Collections. Se trata de un conjunto de interfaces, clases y algoritmos para trabajar con una amplia gama de estructuras de datos dinámicas. Hemos aprendido a utilizar conjuntos, listas y árboles, algoritmos para su manipulación e iteradores para su recorrido de forma fácil y eficiente. Para su uso, es importante conocer el concepto de tipos genéricos.





Ministerio de Educación y FP [\(CC BY-NC\)](#)

En la siguiente Unidad nos centraremos en los mecanismos propuestos por el API Java para el intercambio de datos con memoria secundaria.