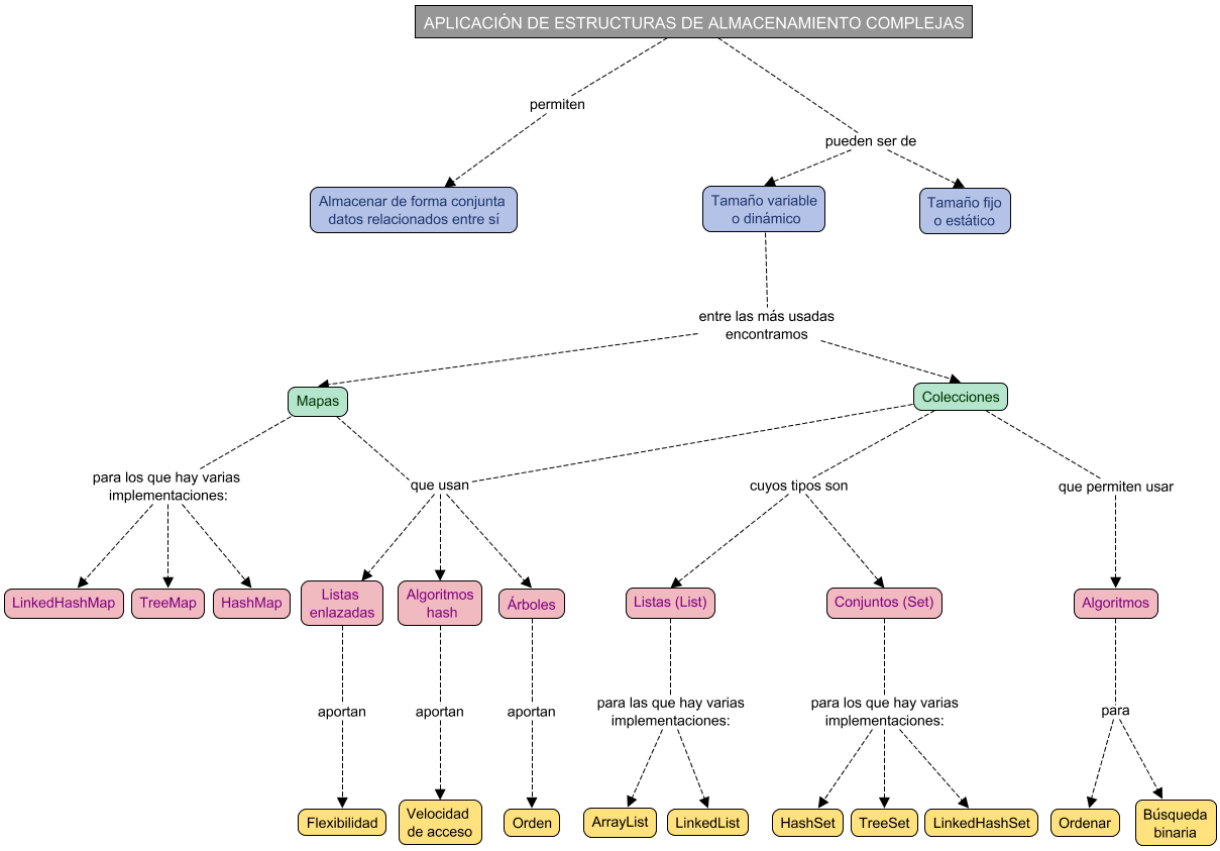
Teoría Internas

https://bronze-infinity-741.notion.site/Teor-a-Internas-7dfac200720e4ece8e658badda566eb1?pvs=4

▼ Mapa Conceptual



▼ Genéricos

▼ Métodos

```
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;
     }
}
```

▼ Invocación de Métodos

```
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);
```

▼ Clases

```
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;
        }
    }
}
```

```
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]);
}
```

```
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);
```

▼ Conceptos

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.

Profundizando

▼ Colecciones

interfaz `java.util.Collection`

Define las operaciones comunes a todas las colecciones derivadas

- 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.

▼ Conjuntos

La interfaz java.util.Set define cómo deben ser los conjuntos, y extiende la interfaz Collection, aunque no añade ninguna operación nueva.

▼ HashSet

• `java.util.HashSet`. Conjunto que almacena los objetos usando tablas hash, 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).

```
import java.util.HashSet;

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!

```
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`.

▼ 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);
}
```

1 3 4 99

(el resultado sale ordenado por valor)

▼ LinkedHashSet

```
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);
}
```

4 3 1 99

(los valores salen ordenados según el momento de inserción en el conjunto)

▼ Set

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>();
```

▼ Acceso a elementos

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

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

▼ Combinando datos de varias colecciones

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

▼ Ordenando elementos

```
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;
   }
}
```

▼ Listas

▼ Características

- Duplicados
- Acceso posicional
- Búsqueda
- Extracción de sublistas

Dispone de:

- una **interfaz** llamada java.util.List
  - `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).

- y dos **implementaciones** (java.util.LinkedList y java.util.ArrayList)

▼ Uso

```
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.
}
// 2, 3 y 5
```

```
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.
```

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

L**as operaciones aplicadas a una sublista repercuten sobre la lista original**

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

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>()
```

▼ Diferencias entre LinkedList y ArrayList

- 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

▼ Objetos mutables e inmutables

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.

▼ Conjuntos de pares Clave/Valor

Interfaz `java.util.Map` → **Los maps no derivan de la interfaz Collection**

- `java.util.HashMap`
- `java.util.TreeMap`
- `java.util.LinkedHashMap`.

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

▼ Métodos

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>();
```

▼ Iteradores

Un **iterador** se crea invocando el método "**iterator()**" de cualquier **colección**. Veamos un ejemplo (en el ejemplo **t es una colección cualquiera**):

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

→ 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.

▼ 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

▼ Recorrer una Colección

```
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.
}
// La lista después de ejecutar el bucle contendría los números impares.
```

▼ Iterar con Maps

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.

→ 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**.

▼ Algoritmos

▼ ¿Qué podemos hacer con las colecciones?

- **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.**

Ordenación natural en listas y arrays

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);`

▼ Ordenar elementos de una Colección

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

Existen 2 mecanismos

▼ Primera Forma

Crear una clase que implemente la interfaz java.util.Comparator, lo que supone implementar el método compare() definido en dicha interfaz.

```
class ComparadorArticulos implements Comparator<Articulo>{
    @Override
    public int compare(Articulo articulo1, Articulo articulo2) {
       return articulo1.codigoArticulo.compareTo(articulo2.codigoArticulo);
    }
}
```

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

▼ Segunda Forma

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);
    }
}
```

```
Collections.sort(articulos);
```

▼ Operaciones adicionales

`clases java.util.Collections`

y `java.util.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);`

Dividir una cadena en partes.

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