

# UNIDAD 6. ESTRUCTURAS DE ALMACENAMIENTO DINÁMICAS - COLECCIONES Y MAPAS

---

## Índice

- UNIDAD 6. ESTRUCTURAS DE ALMACENAMIENTO DINÁMICAS - COLECCIONES Y MAPAS
  - Índice
  - Introducción
  - Clasificación
  - Las interfaces
  - Tipos de datos parametrizados o genéricos
    - Clases con parámetros genéricos
    - Parámetros genéricos limitados
    - Métodos genéricos
    - Comodines o wildcards
    - Cosas que no se pueden hacer con parámetros genéricos
  - Interface collection
    - Interface List
    - Interface Queue
    - Clase ArrayList
    - Clase LinkedList
    - Interface Set
    - Clase HashSet
    - clase LinkedHashMap
    - Interface SortedSet
      - Estructura en Árbol. Lista ordenada
    - Clase TreeSet
  - Interface Map
    - Clases de la interfaz Map

## Introducción

**Las colecciones** son estructuras dinámicas. Esto quiere decir que pueden aumentar o disminuir su tamaño dependiendo de los elementos que almacenan en tiempo de ejecución.

Mejora respecto a las estructuras de datos estáticas: se altera el tamaño definido en la creación y se puede alterar en tiempo de ejecución.

Todo lo que se almacena en las colecciones se denomina elemento y estos elementos pueden ser añadidos, eliminados, recorridos, evitar duplicados u ordenados dependiendo del tipo de colección que utilicemos.

La librería de Java Collections Framework (JCF) (**java.util**) está compuesta de:

- **Colecciones:** Interfaces que identifica una colección de objetos independiente de la implementación
- **Contenedores:** Implementaciones de colecciones. Serán contenedores ArrayList, HashSet, LinkedList, etc. Se implementarán las colecciones ordenadas, desordenadas, sin duplicidad de elementos o sin ella.

- **Algoritmos:** que trabajan sobre las colecciones, de un modo polimórfico.

El número de interfaces, clases abstractas y clases disponibles para la creación de colecciones es espectacular. Poco a poco iremos viendo las principales.

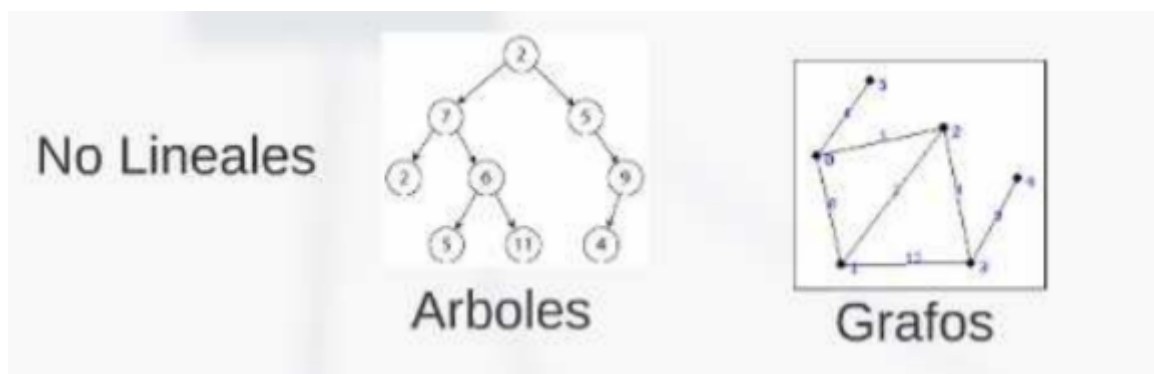
## Clasificación

La clasificación en dos grupos:

- Lineales

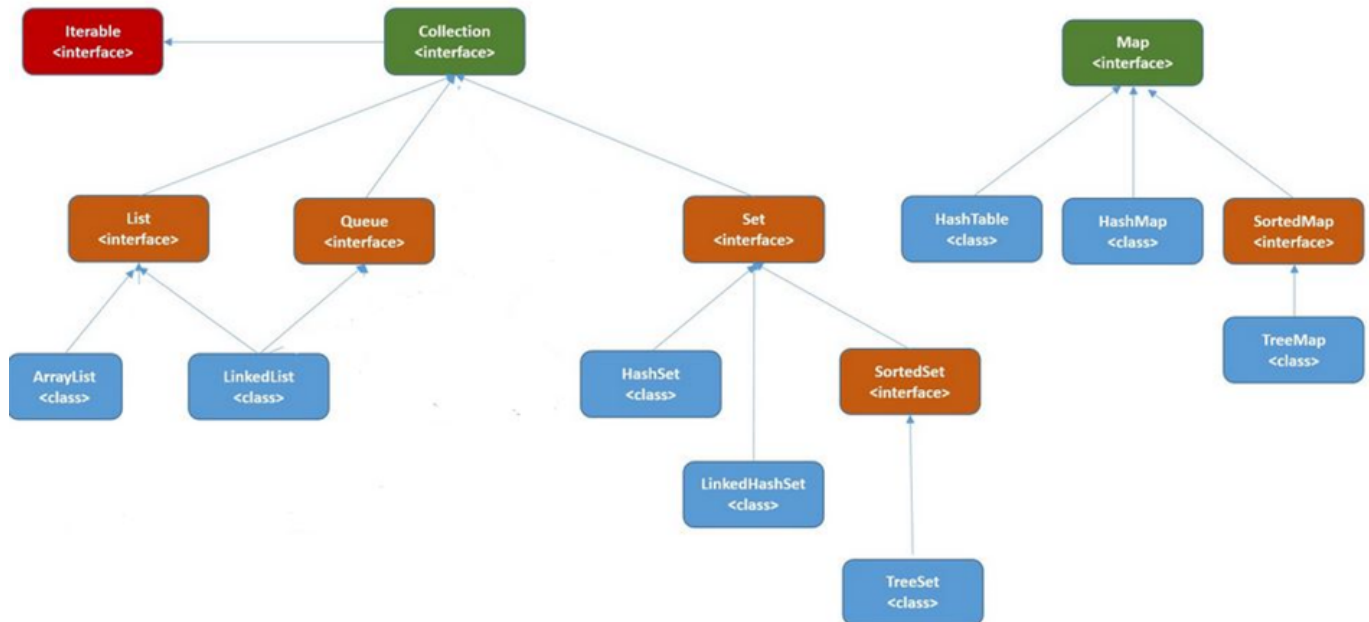


- No lineales



## Las interfaces

**Las estructuras dinámicas** se implementan **por medio de interfaces**. Una interfaz en java es una colección de métodos abstractos y propiedades constantes. En las interfaces se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.



## Tipos de datos parametrizados o genéricos

**Las colecciones** trabajan con tipos de **datos genéricos**.

El uso de los tipos genéricos obedece a la necesidad de disponer de clases, interfaces o métodos que se puedan usar con muchos tipos de datos distintos, pero haciendo **comprobaciones de tipo en tiempo de compilación**.

Ejemplos son los métodos de comparación, `compareTo()` y `compare()`, que aparecen en las interfaces `Comparable` y `Comparator` respectivamente.

Ambas interfaces están pensadas para comparar objetos de cualquier clase.

Implementamos `Comparable` para cualquier clase de objetos que insertemos en cualquier colección que pretendamos ordenar.

**Implementando genéricos** si hay **error** se manifiesta durante **la compilación**:

```

class Estudiante implements Comparable<Estudiante>{

    public int compareTo( Estudiante o){
        return this.nombre.compareTo(o.nombre);
    }
}
  
```

**Sin implementar parámetros genéricos** si hay **error** con el tipo de dato del parámetro no se manifiesta durante la compilación, sino **en tiempo de ejecución**, lanzando una excepción:

```
class Estudiante implements Comparable{

    public int compareTo( Object o){
        return this.nombre.compareTo(((Estudiante)o).nombre);
    }
}
```

## Clases con parámetros genéricos

Si queremos crear una clase que permita trabajar con todo tipo de objetos, es necesario declarar esa clase como genérica. Veremos con un ejemplo como se implementa esta genericidad en Java.

Ejemplo: una clase Contenedor que sirva para todo tipo de objetos y que a su vez permita controlar en cada caso ese tipo => **Una clase contenedor con tipo genérico T**

**T representa el tipo de datos** que va usar la clase en cada declaración concreta, y **tiene que ser clase o interfaz, nunca un tipo primitivo**.


```
public class Contenedor<T>{
    // atributo
    private T objeto;
    // se inicializa a null contenedor vacío
    public Contenedor(){

    }
    // agregar un objeto
    public void guardar (T nuevo){
        objeto=nuevo;
    }
    // sacar el objeto
    public T extraer(){
        T res=objeto;
        // el contenedor vuelve a estar vacío
        objeto=null;
        return res;
    }
}
```

Podemos utilizarlo para crear un Contenedor para enteros, para numeros reales o de clientes. La sintaxis es:

```
//contenedor de enteros
Contenedor<Integer> contenedor1= new Contenedor<Integer>();
//otra sintaxis
Contenedor<Integer> contenedor2= new Contenedor<>();
// contenedor de reales
Contenedor<Double> contenedor3= new Contenedor<>();
```

```
//contenedor de clientes, partimos que tenemos una clase cliente
Contenedor<Cliente> contenedor4= new Contenedor<>();
```

 Hoja de ejemplos (EjemploGenerico)

[Enlace a la API Java 18. Interfaz Comparable](#)

[Enlace a la API Java 18. Interfaz Comparator](#)

En general, se usa la letra T para el tipo genérico, U para arrays, E para elementos de colecciones, K para claves, V para valores o N para números.

Ya hemos comentado la interfaz Comparable de java. Otra interfaz que ha sido redefinida con tipos genéricos es **Comparator**.

Ejemplo: Implementar una clase comparadora para ordenar los estudiantes por orden alfabético de nombres.

```
public class ComparaNombres implements Comparator<Estudiante>{
    @override
    public int compare(Estudiante e1, Estudiante e2){
        return e1.getNombre().compareTo(e2.getNombre());
    }
}
```

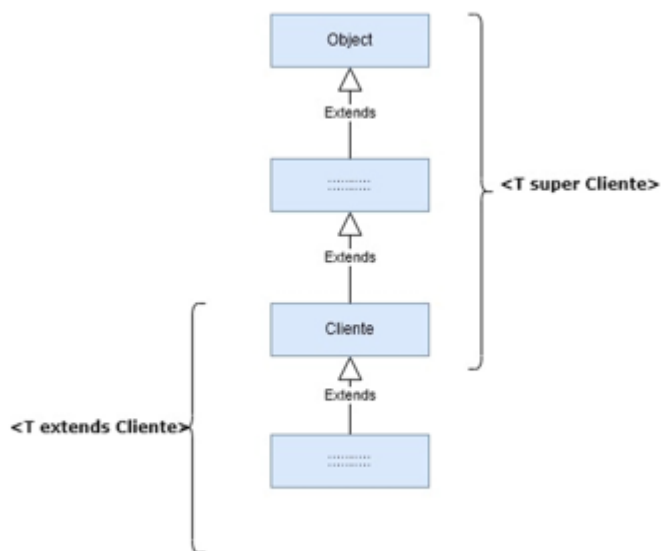
Ejemplos de interfaces de java implementadas con genéricos

```
public interface Comparable<T>{
    int compareTo(T o);
}
public interface Comparator<T>{
    int compare(T o1, T o2);
}
```

## Parámetros genéricos limitados

A veces las operaciones solo tienen sentido para determinados tipos de T. Por ejemplo, si queremos hacer una operación aritmética entre valores de tipo T, entonces no puede ser String, ni Estudiante.

La idea es que se limiten los posibles tipos de T a una determinada clase claseLimite y todas sus subclases ( si claseLimite es un límite superior) o todas sus superclases( si claseLimite es un límite inferior)



- Si claseLimite es un límite superior, definiremos:

```
class nombreClase <T extends claseLimite>
```

Significa que T puede ser claseLimite o cualquiera de sus subclases

- Si clase claseLimite es un limite inferior, definiremos:

```
class nombreClase <T super claseLimite>
```

Significa claseLimite y todas sus superclases

## Métodos genéricos

Normalmente los métodos genéricos aparecen implementados dentro de clases o interfaces genéricas. Sin embargo, dentro de cualquier clase, podemos implementar métodos con sus propios parámetros genéricos.

Ejemplo, vamos a implementar un método que nos devuelve el número de elementos null que hay en un vector que se le pasa como argumento.

El tipo U del vector es genérico, y se declara en la definición del método, justo antes del tipo de dato devuelto.

```
public static <U> int numeroDeNulos( U[] vector){
    int cont=0;
    for (U elemento: vector){
        if (elemento==null){
            cont++
        }
    }
    return cont;
}
```

## Comodines o wildcards

Los comodines se suelen usar en la declaración de atributos, variables locales o parámetros pasados a un método.

**Un comodín se representa con un símbolo ? Significa cualquier tipo**

Ejemplo

```
Contenedor<?> contenedor;
```

Hemos declarado una variable contenedor de tipo Contenedor cuyo parámetro genérico asociado puede ser cualquiera.

Esto significa que todos los objetos Contenedor pertenecen a alguna subclase de Contenedor<?>, como por ejemplo un contenedor de enteros o contenedor de cliente declarados anteriormente.

Un comodín también se puede usar en la clase limite superior

```
<? extends T>;
```

Significa cualquier clase que herede de T, incluyendo a esta.

Un comodín para limitar una clase limite inferior

```
<? super T>;
```

Significa T o cualquier superclase de T

## Ejemplo

```
Contenedor <? extends Number> contenedor;
```

Estamos declarando una variable de tipo Contenedor de tipo numérica, es decir, puede referenciar a un objeto de tipo Integer, o de tipo Double, etc...Todas son subclases de Number.

## Cosas que no se pueden hacer con parámetros genéricos

1. Los tipos genéricos nunca pueden ser primitivos. Cuando nos hagan falta, usaremos las clases envoltorio Integer, Character ...
2. No se pueden crear instancias de tipo genérico, como en `new T()`;
3. No se pueden crear arrays de tipos genéricos, como en `new T[10]`. Cuando se necesiten se pasan como argumento al método donde van a ser usadas.
4. Tampoco se pueden crear tablas de clases parametrizadas, como `new Contenedor[5]`;
5. No se pueden usar excepciones genéricas.



Hoja de ejercicios 1 genéricos

## Interface collection

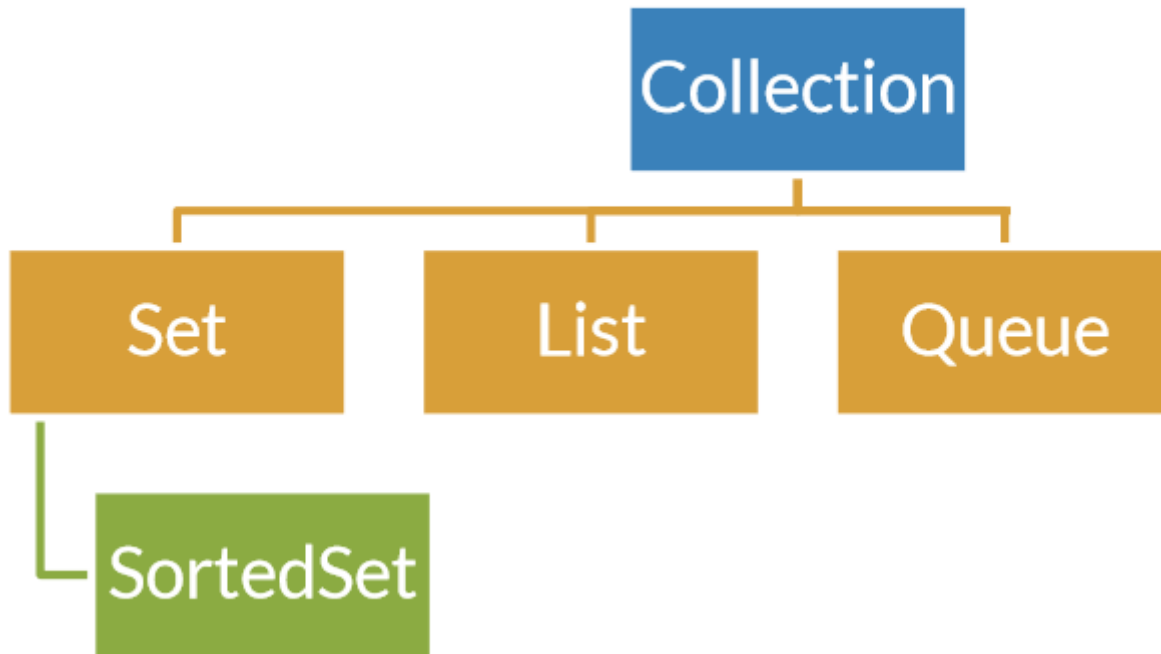
La interface fundamental de trabajo con estructuras dinámicas es `java.util.Collection`. Es la raíz del funcionamiento de las colecciones y representa objetos que tienen la capacidad de almacenar listas de otros objetos.

Están implementadas con dos versiones

- Utilizando **Genericidad**: `Collection<E>`
- Utilizando **Polimorfismo**: `Collection`

La interfaz principal es `Collection`, y tiene a su vez tres interfaces hijas. Cada una de ellas representa distintos tipos de colecciones que funcionan de determinada manera.





- Las listas son lineales, hay posibilidad de ordenarlas, y sus elementos pueden estar repetidos.
- Los conjuntos (set) no soportan elementos repetidos y hay posibilidad de ordenar sus elementos.
- Las colas (queue) funcionan como su nombre indica, el primer elemento que llega es el primer elemento que sale, (como en la cola del cine).

Los métodos más importantes de la interfaz Collection son los siguientes:

Nombre	Uso
size	Devuelve el número de elementos
add	Inserta un objeto. Devuelve true si lo inserta
contains	Indica si un objeto pertenece a la colección
remove	Elimina una referencia del objeto si existe. Devuelve true si lo realiza
clear	Limpia todas las referencias
iterator	Devuelve una implementación de un iterado
addAll	Añade todos los elementos de la colección pasada como argumento
containsAll	Indica si todos los objetos pertenecen a la colección
removeAll	Elimina todas las referencias a los objetos si existiesen
retainAll	Elimina todas las referencias a los objetos no pasados

Existen dos modos para recorrer las colecciones:

- Bucle for-each

```
for( String elemento:collection){  
    system.out.println(elemento);  
}
```

- Mediante iteradores

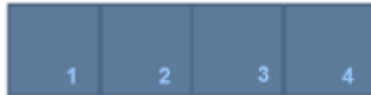
```
Iterator <String> iterator = collection.iterator();  
while (iterator.hasNext()){  
    String cadena=it.next();  
    system.out.println(cadena);  
}
```



Hoja de ejemplos (EjemploCollection)

## Interface List

- Los elementos tienen posición
- Permite **duplicados**
- También permite búsqueda e iteraciones
- Las implementaciones más conocidas son **ArrayList y LinkedList**.
- Si no sabemos cual escoger, utilizaremos siempre ArrayList.



## *List*

- Lineal
- Posibilidad de orden.
- Con repetidos

- Inclusión de los genéricos
- Permiten parametrizar el tipo
- Operador diamond

```
List<String> cars = new ArrayList<String>();  
//Nos ahorra indicar dos veces el tipo A partir de Java 1.7  
List<String> cars = new ArrayList<>();
```

Métodos más importantes de la interface List.

Nombre	Uso
add	Inserta un objeto. Devuelve true si lo inserta
remove	Elimina una referencia del objeto si existe. Devuelve true si lo realiza
set	Sustituye el elemento número índice por uno nuevo. Devuelve además el elemento antiguo
get	Obtiene el elemento almacenado en la colección en la posición que indica el índice
indexOf	Devuelve la posición del elemento. Si no lo encuentra, devuelve -1
lastIndexOf	devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1



Hoja de ejemplos (EjemploList)

## Interface Queue

- Una cola es una estructura de datos de tipo FIFO (First input, First Output)
- Una cola está diseñado para que los elementos sean insertados al final de la cola, y los elementos eliminados sean los del principio de la cola.
- se implementa para gestionarla con la clase **LinkedList**

```
//Declarar una Queue:  
Queue<nombreClase> nombreCola;  
//Crear una Queue :  
nombreCola = new LinkedList();  
//Como suele ser habitual, se puede declarar la lista a la vez que se crea:  
Queue<nombreClase> nombreCola = new LinkedList();
```

Los métodos más importantes de la interfaz queue son los siguientes:

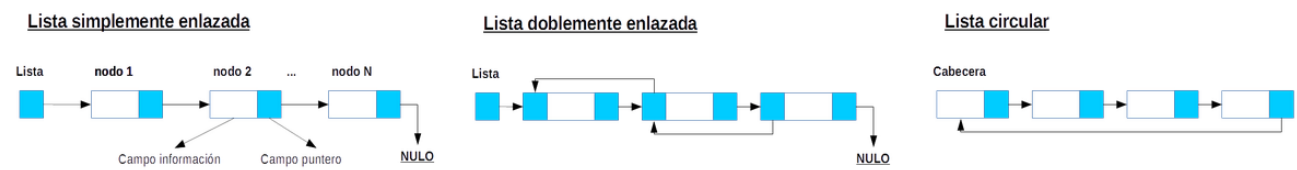
Nombre	Uso
add	Inserta elemento en la cola. Devuelve true si lo inserta
peek	Devuelve el elemento pero no borra la cabeza de la cola
element	Como la anterior pero lanza una excepción si la cola esta vacía
poll	Devuelve y elimina la cabeza de la cola
remove	Como al anterior pero lanza una excepción si está vacía

## Clase ArrayList

Implementa **la interface List** y posee 3 constructores:

```
//constructor por defecto, crea un ArrayList vacío
ArrayList();
//Crea una lista con una capacidad inicial indicada por parámetro
ArrayList(int capacidadInicial)
//Crea una lista a partir de los elementos de la colección pasada por parámetro
ArrayList( Collection| Collection<E> collection)
```

Nos permitirá implimentar cualquier tipo de listas



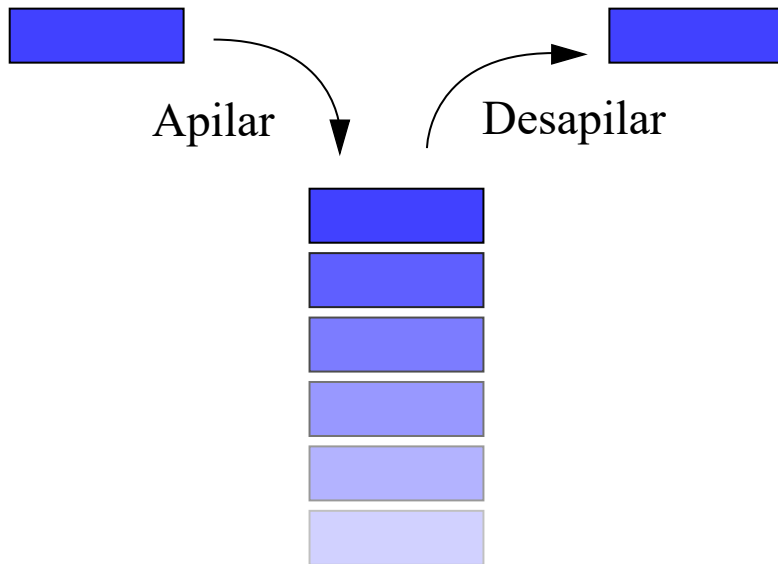
## Clase LinkedList

Implementa **las interfaces List y Queue**. Tiene métodos que permiten crear listas de adición tanto por delante como por detrás (listas dobles). Desde esta clase es sencillo implementar estructuras en forma de pila o de cola.

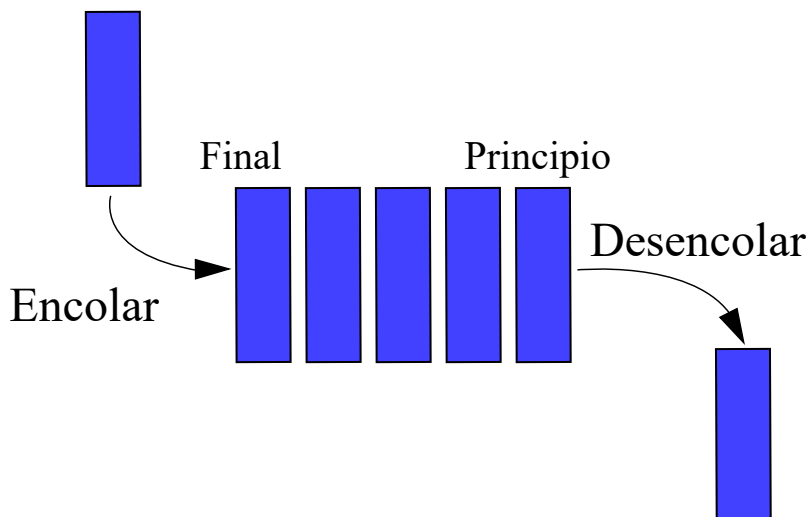
Nombre	Uso
getFirst	Obtiene el primer elemento de la lista
getLast	Obtiene el último elemento de la lista
addFirst	Añade el objeto al principio de la lista
addLast	Añade el objeto al final de la lista
removeFirst	Borra el primer elemento
removeLast	Borra el último elemento


Los métodos están pensados para que las listas creadas mediante objetos **LinkedList** sirven para añadir elementos por la cabeza o la cola, pero en ningún caso por el interior.

En el caso de **implementar pilas**, los nuevos elementos de la pila se añadirían por la cola (mediante **addLast**) y se obtendrían por la propia cola (**getLast, removeLast**).



En el caso de **implementar colas**, los nuevos elementos se añaden por la cola, pero se obtienen por la cabeza (**getFirst**, **removeFirst**).



 Hoja de ejemplos (EjemploListasOrdenadas)

 Hoja de ejercicios de colecciones 1

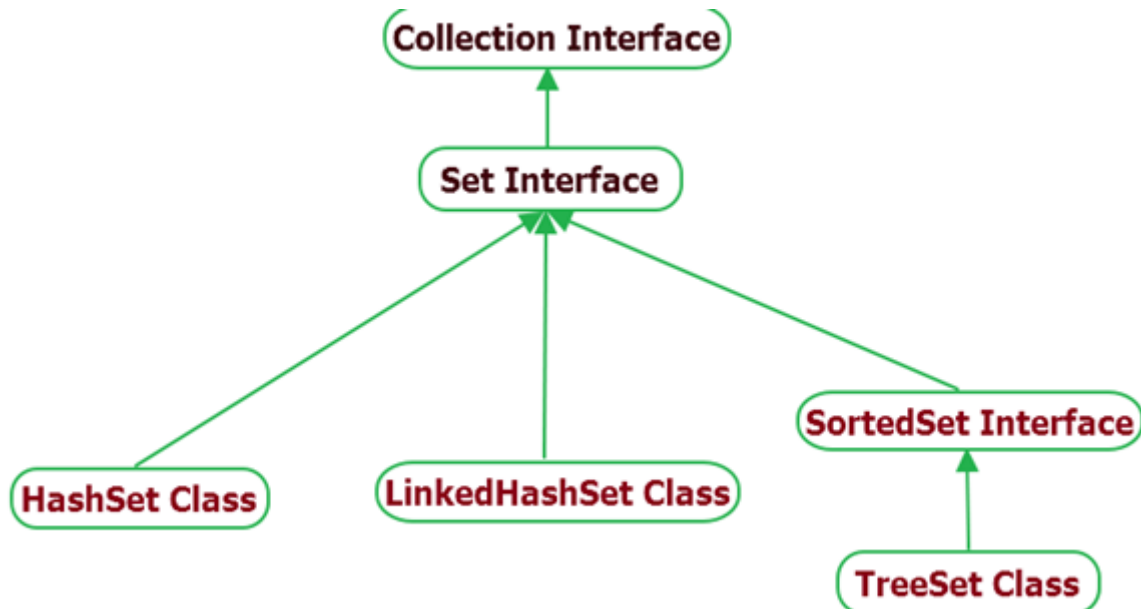
 Hoja de ejercicios de colecciones 2

## Interface Set

La interfaz **Set** representa una repetición de elementos que **no pueden estar duplicados**

Hereda de la interface Collection y, por tanto, posee los mismos métodos. La diferencia está en el uso de duplicados.

Es el **método equals** el que se encarga de determinar **si dos objetos son duplicados** en la lista (habrá que redefinir este método para que funcione adecuadamente).



Listas:

- **HashSet**: Conjunto desordenado
- **LinkedHashSet**: Conjunto desordenado que guarda el orden de inserción (mediante una lista enlazada)
- **TreeSet**: Conjunto ordenado

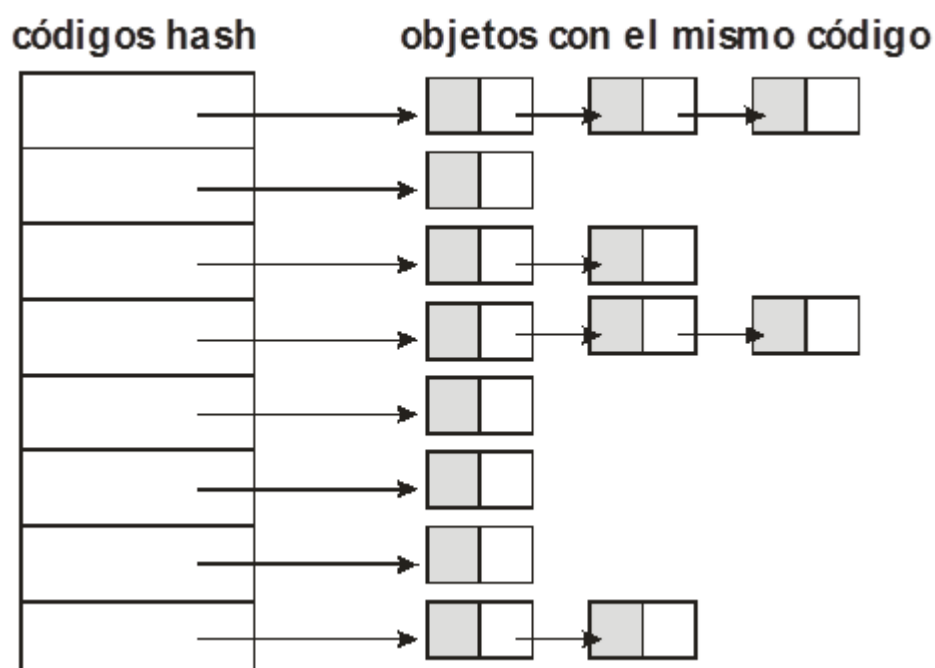
## Clase HashSet

Implementa **listas sin duplicados**.

Tiene los métodos de la **interfaz Collection**

**Internamente una tabla de tipo hash.** Esas tablas asocian claves a conjuntos de valores.

La naturaleza de las tablas hash hace que cuando se crean listas HashSet, no habrá valores duplicados, pero en absoluto se garantiza el orden. Es decir, cada vez que llega un valor único al array se añade en una posición del mismo (la siguiente que esté libre), si llega otro con el mismo valor se añade a la lista de esa celda del array.



Pero ¿cómo compara Java los objetos de la lista para saber si son iguales?

- Utiliza el **método equals** heredado de la clase base Object. Por ello es necesario que las clases de los objetos que se almacenarán en la tabla definan ese método.
- Definir también el método heredado **hashCode**. La razón es que es el hashCode es el código pensado para este tipo de lista, de hecho es el identificador en una lista HashSet, por ello los objetos que consideremos iguales en contenido deben devolver el mismo hashCode, es decir, el mismo número entero.

## clase LinkedHashSet

Se trata de una clase heredera de la anterior con los mismos métodos y funciones, pero que consigue mantener el **orden en el que los datos fueron insertados**.

Sigue necesitando **hashCode y equals** en la clase.



Hoja de ejemplos (EjemploHashSetYLinkedHashSet)



Hoja de ejercicios de colecciones 3

## Interface SortedSet

La interfaz SortedSet es la encargada de definir **una estructura en arbol**.

Esta interfaz deriva de **Collection** y añade estos métodos:

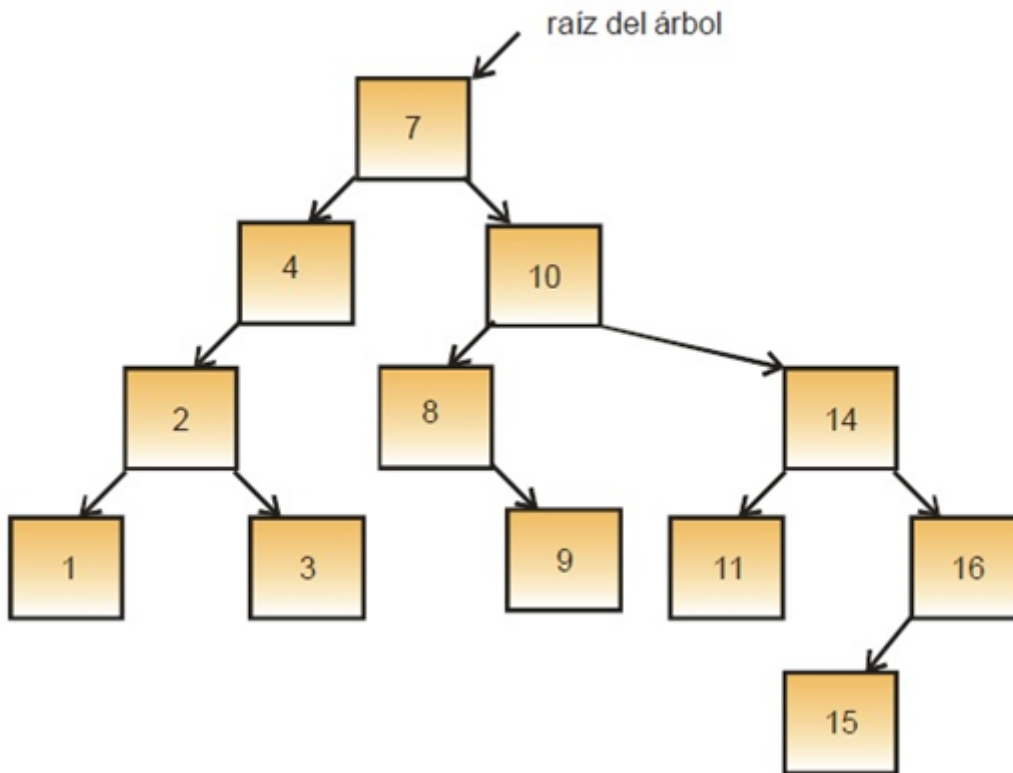
Nombre	Uso
first	Obtiene el primer elemento del árbol(el más pequeño)
last	Obtiene el último elemento del árbol(el más grande)
headSet	Obtiene un SortedSet que contendrá todos los elementos menores que el objeto o
tailSet	Obtiene un SortedSet que contendrá todos los elementos mayores que el objeto o
subSet	Devuelve la posición del elemento en el árbol (-1 significa que no lo ha encontrado)

## Estructura en Árbol. Lista ordenada

Un árbol es una estructura en la que los datos se organizan en nodos los cuales se relacionan con dos o más nodos. En general se utilizan para ordenar datos y en ese caso de cada nodo sólo pueden colgar otros dos de modo que a la izquierda cuelgan valores menores y a la derecha valores mayores.

Al recorrer esta estructura, los datos aparecen automáticamente en el orden correcto. La adición de elementos es más lenta, pero su recorrido ordenado es mucho más eficiente.





## Clase TreeSet

Se trata de la clase que se utiliza prioritariamente para conseguir **árboles ordenados** ya que implementa la **interfaz SortedSet**.

El problema es que los objetos tienen que poder ser comparados para determinar su orden en el árbol. Esto implica implementar **la interfaz Comparable** de Java (está en java.lang).

Esta interfaz define el **método compareTo** que utiliza como argumento un objeto a comparar y que devuelve 0 si los objetos son iguales, un número positivo si el primero es mayor que el segundo y negativo en caso contrario.

Con lo cual **los objetos a incluir en un TreeSet deben implementar Comparable y esto les obliga a redefinir el método compareTo**



Hoja de ejemplos (EjemploTreeSet)



Hoja de ejercicios de colecciones 4

# Interface Map

Representa una estructura de datos para almacenar pares "clave/valor".

Para cada clave, solo tiene un valor.

principales métodos

```
nombreMap.size();//Devuelve el número de elementos del Map
nombreMap.isEmpty();// Devuelve true si no hay elementos en el Map y false si si
los hay
nombreMap.put(K clave, V valor); //Añade un elemento al Map
nombreMap.get(K clave);//Devuelve el valor de la clave que se pasa como parámetro
o null si la clave no existe
nombreMap.clear();//Borra todos los componentes del Map
nombreMap.remove(K clave);//Borra el par clave/valor de la clave que se pasa como
parámetro
nombreMap.containsKey( K clave);//Devuelve true si en el Map hay una clave que
coincide con K
nombreMap.ContainsValue(V valor); //Devuelve true si en el Map hay un valor que
coincide con V
nombreMap.values();// Devuelve una colección con los valores el Map
```

## Clases de la interfaz Map

- **HashMap:** Los elementos que inserta en el map no tendrán un orden específico. No aceptan claves duplicadas ni valores nulos.
- **TreeMap:** El Mapa lo ordena de forma "natural". Por ejemplo, si la clave son valores enteros, los ordena de menor a mayor.
- **LinkedHashMap:** Inserta en el Map los elementos en el orden en el que se van insertando; es decir, que no tiene una ordenación de los elementos como tal, por lo que esta clase realiza las búsquedas de los elementos de forma más lenta que las demás clases.



Hoja de ejemplos (EjemploMap)



Hoja de ejercicios de colecciones 5



Hoja de ejercicios de colecciones 6 repaso