Colecciones en Java

Contenido

1.- Introducción

Java tiene desde la versión 1.2 todo un juego de clases e interfaces para guardar colecciones de objetos. En él, todas las entidades conceptuales están representadas por interfaces, y las clases se usan para proveer implementaciones de esas interfaces.

Como corresponde a un lenguaje orientado a objetos, estas clases e interfaces están estructuradas en una jerarquía. A medida que se va descendiendo a niveles más específicos aumentan los requerimientos y lo que se le pide a ese objeto que sepa hacer.

La mayoría de lo que llamamos colecciones descienden de dos interfaces genéricas

- **Collection**: Lista de elementos (distintos entre sí o no). Permite guardar elementos de una clase genérica E.

  Definición: Interface Collection<E>

  ```
  E - the type of elements in this collection
  ```

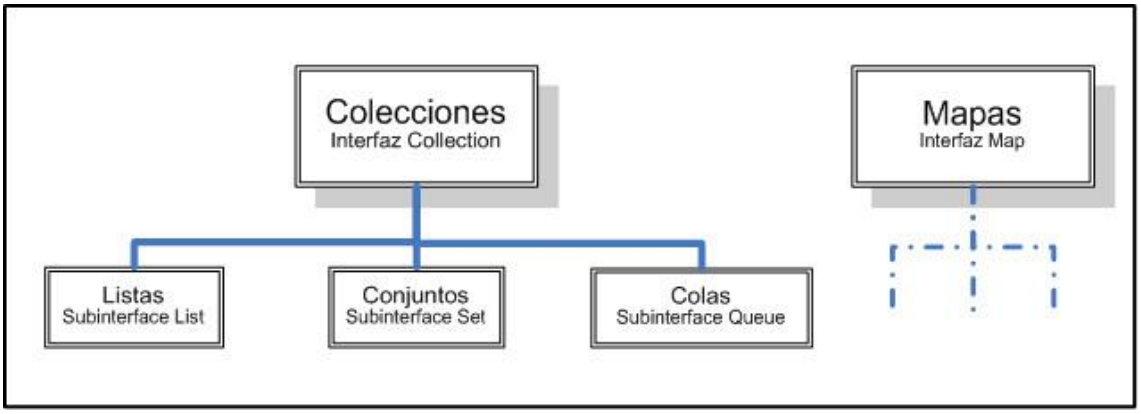
- **Map**: Guarda la correspondencia entre claves únicas y valores (diccionarios). Permite acceder a los elementos de la clase V por el valor de una clave perteneciente a la clase K (como alternativa al acceso por un índice entero). En otros lenguajes se conocen como “arrays asociativos”.

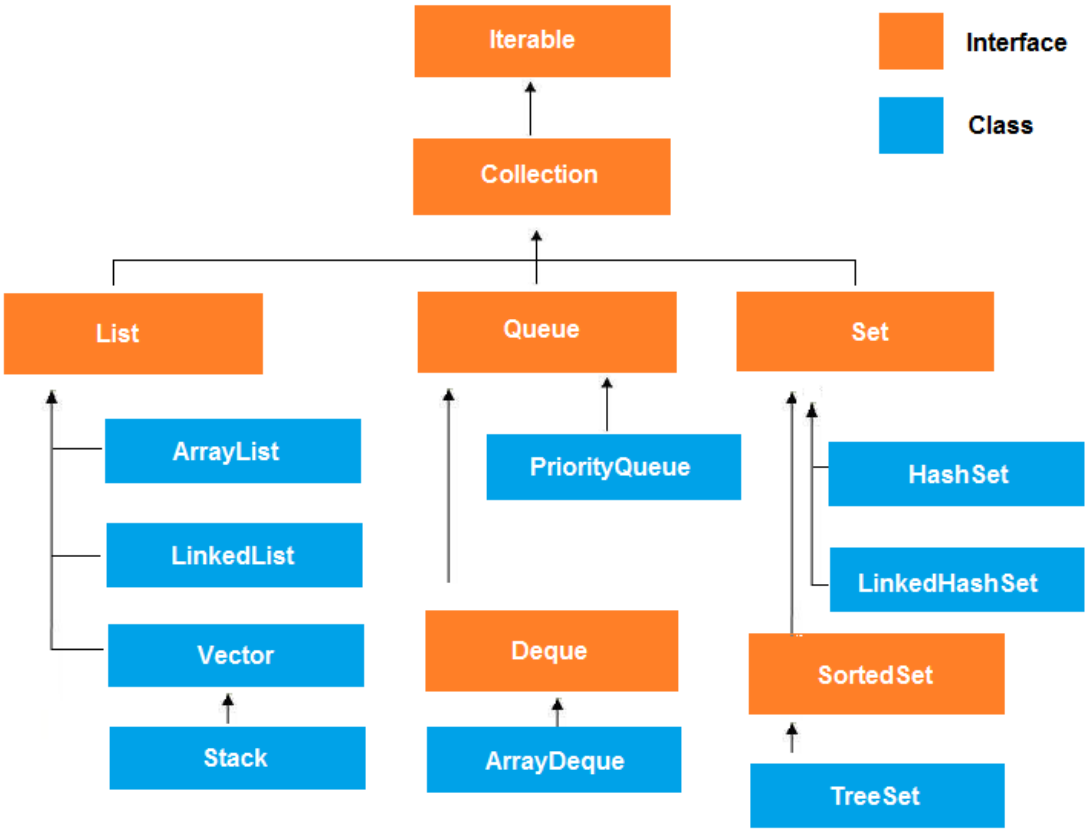
  Definición: Interface Map<K,V>

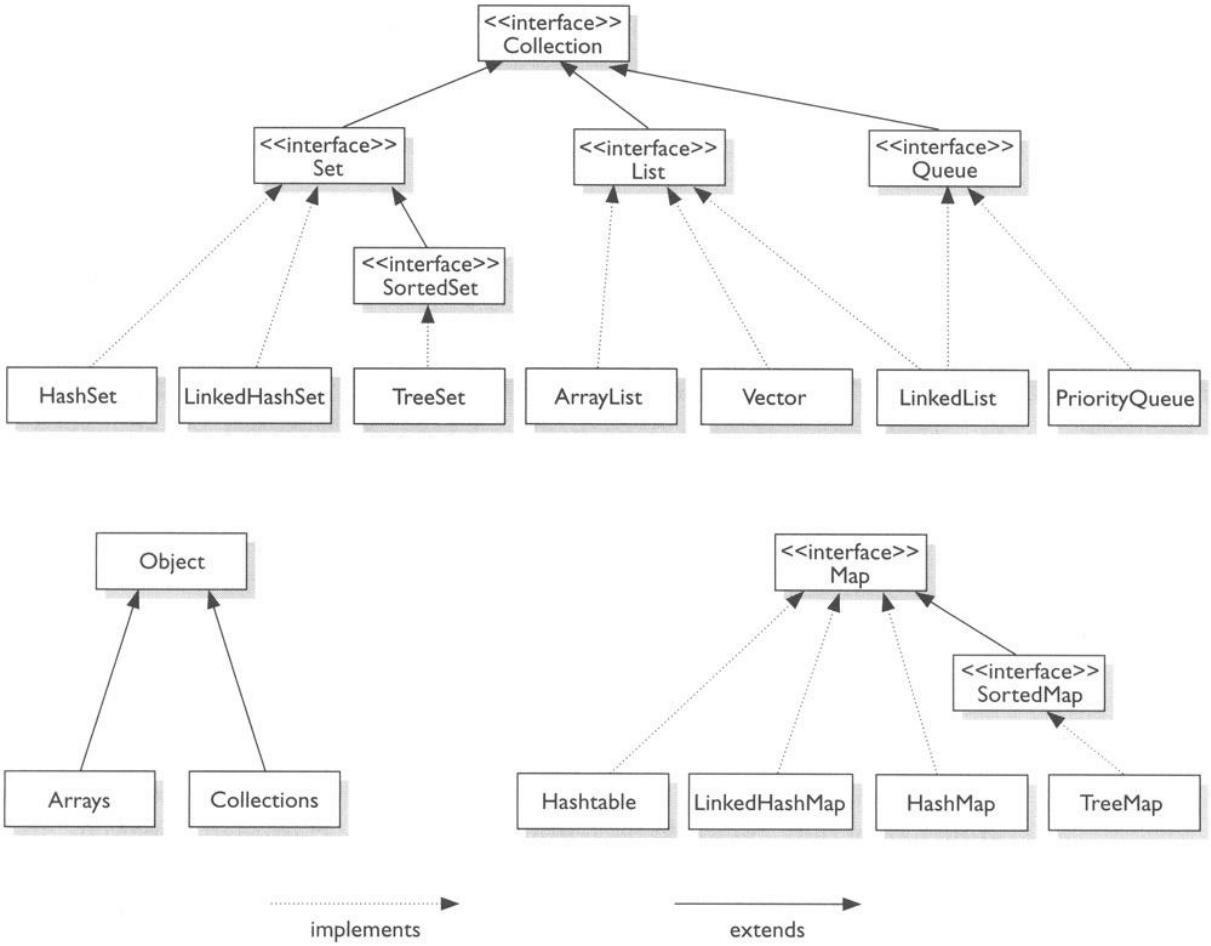
  ```
  K - the type of keys maintained by this map
  V - the type of mapped values
  ```

Java no implementa directamente ninguna de esas interfaces. Las clases que lo hagan deben proporcionar dos constructores:

- **Constructor vacío** (crea una colección/mapa sin elementos)
- **Constructor con un argumento de tipo Collection/Map** (crea una copia de la colección/mapa pasado como argumento)


Colecciones
Interfaz Collection
Mapas
Interfaz Map
Listas
Subinterface List
Conjuntos
Subinterface Set
Colas
Subinterface Queue


Iterable
Collection
List
Queue
Set
ArrayList
LinkedList
Vector
Stack
PriorityQueue
Deque
ArrayDeque
HashSet
LinkedHashSet
SortedSet
TreeSet
Interface
Class



2.- Interfaz Collection<>

La interfaz más importante es **Collection**. Una Collection es todo aquello que se puede recorrer (o "iterar") y de lo que se puede saber el tamaño. Muchas otras clases extenderán Collection imponiendo más restricciones y dando más funcionalidades.

No se puede hacer "new" de una Collection, sino que todas las clases que realmente manejan colecciones "son" Collection, y admiten sus operaciones.

Las operaciones básicas de una Collection son:

MÉTODO	TIPO DEVUELTO	PROPÓSITO
add(T e)	boolean	Trata de agregar un nuevo elemento al final de la colección. Devuelve true si lo consigue
addAll(Collection e)	boolean	Trata de agregar al final de la colección cada elemento de la colección de elementos e. Devuelve true si lo consigue
clear()	void	Elimina todos los elementos de

		la colección
contains(Object o)	boolean	Devuelve true si el objeto pasado forma parte de la colección
containsAll(Collection col)	boolean	Devuelve true si todos y cada uno de los objetos de la colección col forman parte de la colección
isEmpty()	boolean	Devuelve true si la colección está vacía
iterator()	Iterator	Genera un nuevo objeto iterador a partir de una colección
remove(Object o)	boolean	Elimina de la colección una sola instancia de o -si está presente
removeAll(Collection col)	boolean	Elimina de la colección todos los objetos de la colección col
size()	int	Número de elementos de la colección
toArray()	Object[ ]	Genera un array estático con los elementos de la colección Atención: Devuelve un array genérico de Object[ ] -por lo que habrá que realizar después las conversiones necesarias
toArray(T[] a)	T[ ]	Trata de guardar en el array a todos los elementos de la colección. Si la colección no "cabe" en a, crea un nuevo array de tipo T[] con espacio suficiente para almacenarlos

El siguiente ejemplo recorre una colección de Integer borrando todos los ceros:

```
void borrarCeros(Collection<Integer> ceros)
{
    Iterator<Integer> it = ceros.iterator();
    while(it.hasNext())
    {
            int i = it.next();
            if(i == 0)
                    it.remove();
    }
}
```

A partir de Java 6 hay una manera simplificada de recorrer una collection (que sirve si no necesitamos borrar elementos). Se hace mediante un nuevo uso del keyword **for**:

```
void mostrar(Collection<?> col)
{
    for(Object o : col)
            System.out.println(o);
}
```

Principales descendientes de Collection. Son asimismo interfaces que heredan de Collection.:

- **List (listas)**: Colección de elementos (únicos o no) a los que se accede mediante un índice numérico
- **Set (conjuntos)**: Colección de elementos no repetidos.
- **Queue (colas)**: Colección de elementos que son agregados en un extremo y eliminados en otro extremo (este último, no necesariamente diferente del extremo de agregación)

Todos los descendientes de Collection implementan además la interfaz Iterable. Esta interfaz permite recorrerlos mediante un objeto de tipo Iterator o mediante el bucle for -each.

3.- Interfaz List<>

- Esta interfaz define una colección de objetos a los que se accede por un índice (el cual determinará su posición en la lista). Permite guardar elementos duplicados.
- Carecen de métodos para operar en un rango de elementos de la lista. En su lugar, permiten crear una "vista" de un subconjunto de sus elementos mediante sublist().
- Las listas están pensadas para el recorrido secuencial de sus elementos (operaciones cuyo rendimiento es del orden de O(1) / O(n)). NO son eficientes para la búsqueda o acceso aleatorio a un determinado elemento, cuyo coste suele ser O(n2).

Principales **implementaciones** (clases) de la **interfaz List**:

- **Vector**: La más antigua. Incluye soporte multihilo.
- **ArrayList**: La que nosotros hemos usado, pero carece de soporte multihilo (si más de un hilo va a acceder al mismo objeto ArrayList, debe sincronizarse la operación para evitar inconsistencias.
- **LinkedList**: Es más eficiente que ArrayList para la inserción/eliminación de elementos en posiciones intermedias.

La interfaz List tiene métodos propios que agrega a Collection . Los más relevantes son:

MÉTODO	TIPO DEVUELTO	PROPÓSITO
add(int index, T e)	boolean	Trata de agregar un nuevo elemento a la colección en la posición indicada por "index"
get(int index)	T	Devuelve el elemento en la posición indicada por "index"
indexOf(Object o)	int	Devuelve la posición de la primera ocurrencia del objeto o en la lista o -1 si no se encuentra en la lista
sublist(int inicio, int fin)	List	Sublista con todos los elementos de la lista original con índices comprendidos entre "inicio" (incluido) y "fin" (excluido). Ambas listas quedan ligadas

Además, **LinkedList** contiene **métodos** más flexibles para agregar y eliminar elementos de la lista:

- **add(T o) / addLast (T o):** agrega un nuevo elemento al final de la lista.
- **addFirst (T o) / push (T o):** agrega un nuevo elemento al principio de la lista.
- **add(int posicion, T o):** agrega un nuevo elemento en la posición indicada.
- **getFirst():** devuelve el elemento al comienzo de la lista.
- **getLast():** devuelve el elemento al final de la lista.
- **pop():** devuelve y elimina el primer elemento que se encuentre al principio de la lista.

Recorrido de listas. Se hace con el bucle para colecciones:

```
for (T elemento: lista)
{ }
```

O con un iterador. **Interfaz Iterator:**

Los **iteradores** se pueden definir a partir de cualquier objeto que implemente la interfaz Collection:

Obtención:

```
Iterator it = colección.iterator();
```

Al crearlo, el iterador se encuentra al comienzo en la posición anterior al primer elemento de la lista.

Recorrido:

```
while (it. hasNext()){
        it. next();
}
```

Eliminación del elemento actual:

```
it.remove()
```

Interfaz ListIterator.

Es una extensión de los iteradores anteriores, con el añadido de que permiten recorrido en ambas direcciones. Lo incorporan todas las clases descendientes de List.

Obtención:

ListIterator li = lista. **ListIterator**(int posicion_inicial);

Al crearlo, se le puede indicar el elemento de la lista dónde se sitúa inicialmente. Además de los **métodos** de Iterator, ofrece los siguientes:

- **boolean hasPrevious():** Devuelve Verdadero si hay un elemento anterior.
- **void previous():** Retrocede una posición en la lista.
- **int nextIndex():** devuelve el índice del elemento al que se avanzará con next(), pero sin avanzar de posición.
- **int nextPrevious():** devuelve el índice del elemento al que se retrocederá con previous(), pero sin retroceder.
- **set (T o):** sobrescribe el elemento que se encuentra en la posición actual del iterador.

Para recorrerla en sentido inverso desde la posición inicial, debemos situarlo inicialmente en la posición después del último elemento del iterador.

```
ListIterator<String> li = lista.listIterator(lista.size());
while (li.hasPrevious()) {
        System.out.println(li.previous());
}
```

Métodos estáticos de la clase Collections.

La clase **Collections** es una clase de utilidades. Ofrece métodos estáticos para manipulación de colecciones, de los que los más útiles se resumen en el siguiente cuadro:

Método	Propósito
Collections.fill (lista, elemento:T)	Rellena todos los elementos de la lista con un elemento fijo de la clase T
Collections.max(lista)	Elemento de T "mayor" de la lista (el que se encuentra al final en una ordenación)
Collections.min(lista)	Elemento de T "menor" de la lista (el que se encuentra al comienzo en una ordenación)
Collections.binarySearch (lista, elemento:T)	Busca y devuelve la primera posición donde se encuentra el elemento buscado en la lista. La lista debe encontrarse ordenada. Si no encuentra dicho elemento, devuelve un valor negativo.
Collections.disjoint (lista1, lista2)	Devuelve "True" si las dos listas no poseen ningún elemento en común
Collections.addAll (lista, array: T[])	Agrega al final de la lista los elementos del array pasado como argumento (de objetos de T)
Collections.frequency (lista, elemento)	Número de veces que "elemento" se encuentra en la lista
Collections.sort(lista)	Ordena la lista en orden ascendente según el orden natural definido.

Algunos de estos métodos modifican la colección que se les pase como argumento.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html

Nota: otro método útil, perteneciente a la clase Arrays es:

Arrays.asList(elements)

Crea una lista a partir de los elementos de un array:

```
String[] equipo = { "Amanda", "Loren", "Keith" };

List<String> listaEquipo = Arrays.asList(equipo);
System.out.println("Lista : " + listaEquipo);
```

También podríamos hacerlo así:

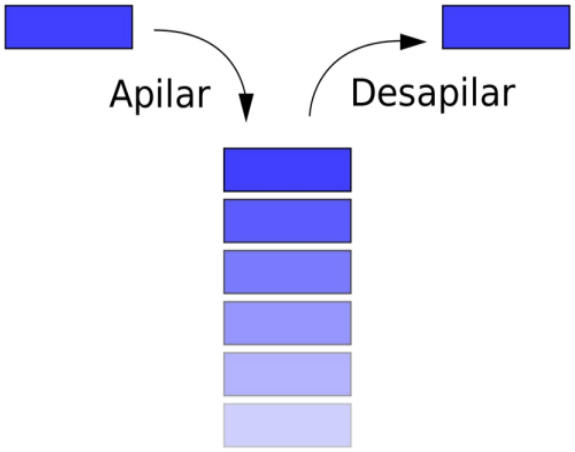
```
List<String> listaEquipo = Arrays.asList("Amanda", "Loren", "Keith");
```

4.- Pilas y Colas

Pila:

Es una estructura dinámica con las siguientes características:

- Está formada por elementos (nodos) pertenecientes a una misma clase T.
- Dichos elementos se disponen secuencialmente.
- Existe un solo extremo activo en los extremos, al que se agregan los nuevos elementos, y del que también se retiran elementos.
- Por ello, cada vez que se retira un elemento, el primer elemento en abandonar la pila es el que llegó el último; por eso, se les conoce como estructuras "**LIFO**" (abreviatura de "Last In, First Out")



En la pila dispondremos al menos de los siguientes **métodos**:

- **Constructor** que instancie la pila vacía.
- Método **apilar (T nodo)**, que inserte un nuevo nodo en el extremo superior de la pila.
- Método **desapilar (): T**, que devuelva el primer nodo del extremo superior (el último en ser insertado) y lo retire de la pila (o devuelva null si la pila está vacía).

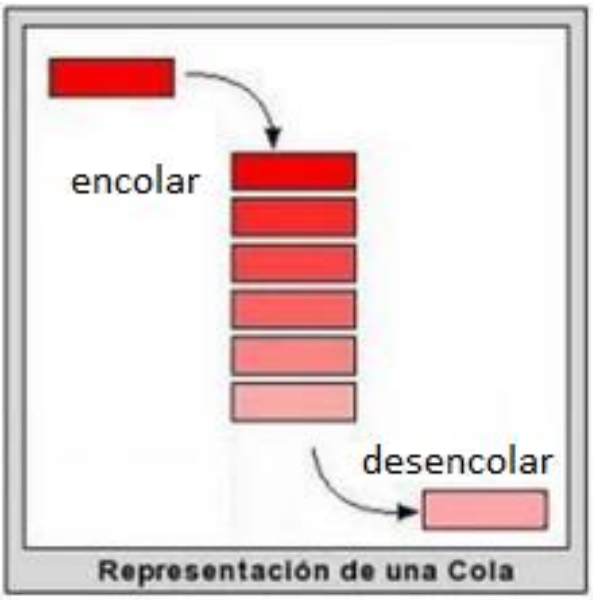
Si utilizamos la colección LinkedList, los métodos push(elemento) y pop() se pueden utilizar para apilar y desapilar, respectivamente.

Cola:

Es una estructura dinámica con las siguientes características:

- Está formada por elementos (nodos) pertenecientes a una misma clase T
- Dichos elementos se disponen secuencialmente.
- Dispone de dos extremos activos opuestos
  - **Cabecera**, en donde se agregan los nuevos elementos.
  - **Cola**, de donde se retiran los elementos.

Entonces, cada vez que se retira un elemento de la cola, sale el nodo más antiguo (el que lleva más tiempo o el primero que se insertó), por eso, se les conoce como estructuras "**FIFO**" (abreviatura de "First In, First Out"),



En la cola dispondremos al menos de los siguientes **métodos**:

- **Constructor** que instancie la pila vacía.
- Método **encolar (T nodo)** que inserte un nuevo nodo en la cabecera
- Método **desencolar (): T** que devuelva el primer nodo de la cola (el primero en ser insertado) y lo elimine de la cola (o devuelva null si la cola está vacía).

Existen varios métodos para desarrollar pilas y colas con LinkedList (buscamos métodos que hacen lo mismo en el jdk). Por ejemplo:

	Pila	**Cola**
Creación	new LinkedList()	new LinkedList()
Agregar(T nodo)	pila **push**(elem)	cola.**addLast**(elem)
Retirar(): T	T elem= pila.pop()	T elem= cola.pop()

5.- Interfaz Set<>

Un Set es una Collection. Set es la palabra inglesa para “conjunto” y los desarrolladores de Java estaban pensando en lo que matemáticamente se conoce como conjunto. Sobre lo que es una Collection, un set agrega una sola restricción: **No puede haber duplicados**.

En un set, generalmente el orden no es dato, ya que la interfaz Set no tiene ninguna funcionalidad para manipularlo (como sí lo admite la interfaz List). Habría que volcar el contenido en una lista y ordenar la lista. O que el set implemente SortedSet y tengamos TreeSet.

La ventaja de utilizar Sets es que preguntar si un elemento ya está contenido mediante “contains()” suele ser muy eficiente. Por ello, **es conveniente utilizarlos cada vez que necesitemos una colección en la que no importe el orden, no se admitan repeticiones del mismo elemento, pero que necesitemos preguntar si un elemento está o no.**

Como el orden no necesariamente es preservado, no existen métodos para “obtener el primer elemento” ni el “último elemento” en la interfaz Set.

Las clases más relevantes son HashSet, TreeSet y LinkedHashSet:

HashSet

Es una clase que guarda internamente sus nodos asociando a cada uno de ellos una **clave única** de acceso, sobre la cual se aplicará una función de direccionamiento o “hash”.

En realidad, es una clase envolvente de una colección de tipo HashMap (que no implementa Collection, ojo). Es decir, internamente es un HashMapa (los Map los veremos más adelante).

Creación:

```
HashSet miTabla= new HashSet<>();
```

También se puede crear copiando en ella los nodos de una lista previamente creada:

```
HashSet miTabla= new HashSet<>(unaLista);
```

Para acceder a cada nodo se usa una función de transformación o “**hash**” que permite calcular la dirección física que corresponde a un valor de la clave.

Cuando introducimos elementos en una colección tipo HashSet, al mostrarnos no tiene por qué mostrarse en ningún orden en particular.

Los Sets (y los Maps) aprovechan que todos los objetos heredan de Object, por lo tanto, todos los métodos de la clase Object están presentes en todos los objetos. Dos de esos métodos son:

- equals(). Para saber si un objeto es igual a otro.
- hashCode(). Se invoca sobre un objeto. Devuelve un número entero de modo que, si dos objetos son iguales, ese número también lo será (se conoce esto como un *hash*).

Para que funcione como debe, si es una clase propia, hay que sobrescribir **equals()** y **hashCode()** en la clase de la que se compone el HashSet. Eclipse es capaz de generarnos los dos métodos.

La clase HashSet usa la función hashCode(). A cada objeto que se añade a la colección se le pide que calcule su "hash". Este valor será un número entre -2147483647 y 2147483648. Basado en ese valor lo guarda en una tabla. Más tarde, cuando se pregunta con contains() si un objeto x ya está, habrá que saber si está en esa tabla. ¿En qué posición de la tabla está? HashSet puede saberlo, ya que, para un objeto determinado, el hash siempre va a tener el mismo valor. Entonces la función contains de HashSet obtiene el hash del objeto que le pasan y va con eso a la tabla. En la posición de la tabla puede haber más de un objeto que tienen ese valor de hash, y si uno de esos es el buscado (equals devuelve true), contains devuelve true.

Un efecto de este algoritmo es que el orden en el que aparecen los objetos al recorrer el set es impredecible. También es importante darse cuenta de que es crítico que la función hashCode() tiene que devolver siempre el mismo valor para los objetos que se consideran iguales (o sea que equals() da true). Si esto no es así, HashSet pondrá al objeto en una posición distinta en la tabla que la que más adelante consultará cuando se llame a contains, y entonces contains dará siempre falso, por más que se haya hecho correctamente el add.

Podemos ver los distintos métodos de que dispone HashSet en el JDK de Java.

TreeSet

Es una clase que guarda internamente sus nodos en un árbol binario.

Una **aclaración** para comprender cómo funciona treeSet. A diferencia de "equals" y "hashCode", que están en todos los objetos, la capacidad de "ordenarse" está solo en aquellos que implementan la **interfaz Comparable**. Cuando un objeto implementa esta interfaz, sabrá compararse con otros (con el método **compareTo()**), y responder con este método si él está antes, después o es igual al objeto que se le pasa como parámetro. Al orden resultante de usar este método se le llama en Java *"orden natural"*.

Muchas de las clases de Java implementan Comparable, por ejemplo, String lo hace, definiendo un orden natural de los strings que es el obvio, el alfabético. También implementan esta interfaz Date, Number, etc. y los "órdenes naturales" que definen estas implementaciones son también los que uno esperaría.

Si yo creo una clase llamada Alumno, queda a mi cargo, si así lo quiero, la definición de un orden natural para los alumnos. Puedo elegir usar el apellido, el nombre, el número de matrícula, etc. De acuerdo al atributo que elija para definir el orden natural codificaré el método compareTo(). Lo que es importante es que la definición de este método sea compatible con el equals(); esto es que a.equals(b) si y solo si a.compareTo(b) == 0.

TreeSet usa una técnica completamente diferente a la explicada para HashSet. **Construye un árbol con los objetos que se van agregando al conjunto**. Un árbol es una forma en computación de tener un conjunto de cosas todo el tiempo en orden, y permitir que se agreguen más cosas y que el orden se mantenga. Al tener todo en orden TreeSet puede fácilmente saber si un objeto está o no.

Una *ventaja* de TreeSet es que el orden en el que aparecen los elementos al recorrerlos es el orden natural de ellos (los objetos deberán implementar Comparable; si no lo hacen se deberá especificar una función de comparación manualmente). Una *desventaja* es que mantener todo ordenado tiene un costo, y esta clase es un poco menos eficiente que HashSet.

El coste de sus métodos básicos es de tipo O(log(n)) - intermedio entre el coste lineal y el coste constante.

Esta clase desciende de la interfaz **SortedSet**, por lo que cualquier objeto de la misma es una instancia de dicha interfaz (así como de Set y de Collection) y puede utilizar los métodos definidos en dichas interfaces.

Creación:

```
TreeSet miTabla= new TreeSet<>();
```

También se puede crear copiando en ella los nodos de una lista previamente creada:

```
TreeSet miTabla= new TreeSet<>(referencia_a_lista);
```

TreeSet posee varios **métodos** heredados de la interfaz SortedSet, como:

- **miTabla.first()**: Devuelve el objeto de la clase T guardado en el primer nodo.
- **miTabla.last()**: Devuelve el objeto de la clase T guardado en el último nodo.
- **miTabla.ceiling(T e)**: Devuelve la referencia al primer nodo que es "mayor o igual" al objeto pasado "e", o null si no existe un nodo así.
- **miTabla.floor(T e)**: Devuelve la referencia al primer nodo que es "menor o igual" al objeto pasado "e", o null si no existe un nodo así.

- **MiTabla.headSet(T e [, boolean incluido])**: Devuelve un conjunto de tipo SortedSet formado por todos aquellos nodos del árbol que son “menores o iguales” que el objeto pasado. Con el parámetro opcional 'incluido', se indica si se incluyen o no los nodos “iguales” a T
- **MiTabla.tailSet(T elem [, boolean incluido])**: Devuelve un conjunto de tipo SortedSet formado por todos aquellos nodos del árbol que son “mayores o iguales” al elemento pasado. Con el parámetro opcional 'incluido', se indica si se incluyen o no los nodos “iguales” a T.

LinkedHashSet

Esta implementación almacena los elementos en función del orden de inserción. Es decir, mantiene el orden en el que se han insertado los elementos. No veremos más sobre ella, solo un ejemplo.

Hashset vs Treeset en Java

• Velocidad e implementación interna:

Hashset: para operaciones como Búsqueda, Insertar y Eliminar, tiempo constante promedio: O (1).
Hashset es más rápido que TreeSet. Hashset se implementa utilizando una tabla hash (HashMap subyacente).
TreeSet: TreeSet toma O(log n) para buscar, insertar y eliminar, que es más alto que el hashset. Pero TreeSet mantiene datos ordenados.
Además, admite operaciones como headSet(), ceiling(), floor(), etc. Estas operaciones también son O(log n) en TreeSet y no se admiten en el hashset. TreeSet se implementa utilizando un árbol de búsqueda binario balanceado. TreeSet tiene un TreeMap subyacente.

• Orden:

No se ordenan elementos en **HashSet**. **TreeSet** mantiene objetos en orden ordenado definido por el método comparable o comparador en Java. Los elementos de TreeSet se clasifican en orden ascendente de forma predeterminada. Ofrece varios métodos para tratar con el conjunto ordenado como first (), last (), headSet (), tailSet (), etc.

• Objeto nulo:

Hashset permite objeto nulo. **TreeSet** no permite el objeto nulo, esto ocurre porque TreeSet usa el método compareTo () para comparar las claves y compareTo () lanzará java.lang.nullpointerexception.

• **Comparación:**

HashSet utiliza el método equals () para comparar dos objetos en el conjunto y para detectar duplicados. **TreeSet** usa el método compareTo () para el mismo propósito.

Nota: Si equals () y compareTo () no son consistentes, es decir, para dos objetos iguales, iguales deberían devolver verdadero, mientras que compareTo () debería devolver cero. SI no se contempla el caso de devolver 0, esto permitirá duplicados en implementaciones establecidas como TreeSet incumpliendo lo establecido en el interfaz Set.

Cuándo preferimos TreeSet en vez de HashSet

- Cuando queremos elementos no repetidos y ordenados, elegimos TreeSet. Por defecto, el orden es ascendente.
- TreeSet cuando necesitamos hacer operaciones de lectura/escritura frecuentemente, TreeSet es una buena elección.

https://www.geeksforgeeks.org/hashset-vs-treeset-in-java/

6.- Interfaz Map<>

Las clases que derivan de esta interfaz mantienen una asociación entre claves de búsqueda y valores asociados a cada clave; de ahí que sean genéricas, basándose en dos clases:

- La clase K, a la que pertenecen las claves de búsqueda (normalmente será una clase basada en enteros, decimales, cadenas u otros tipos primitivos)
- La clase V, a la que pertenecen los valores asociados a cada clave.

Ordena según compare o compareTo, de la clase

Las ocurrencias de las claves en un Map no pueden repetirse; no ocurre lo mismo con las ocurrencias de los valores, por lo que podemos encontrarnos con 2 tipos de mapas:

- **Mapas en donde las ocurrencias de V son únicas**. Dan lugar a una relación 1:1 entre claves y valores
- **Mapas en donde las ocurrencias de V no son únicas**. Dan lugar a una relación N:1 entre claves y valores

Las principales clases derivadas de la interfaz Map son:

- **HashMap**: Clase que no admite sincronización y, además soporta el valor nulo tanto para claves como para valores. Los elementos que inserta en el map no tendrán un orden específico.
- **HashTable**: Clase que admite sincronización entre hilos y no soporta el valor nulo para claves

No entra examen

- **TreeMap**: Clase que deriva de la interfaz SortedMap y se basa en árboles. Mantiene los nodos **ordenados por el valor de su clave**.
- **LinkedHashMap:** Esta implementación almacena las claves en función del orden de inserción. Es un poco más costosa que HashMap. No la vemos.

Recuerda que la interfaz Map está desconectada de la interfaz Collection, por lo que los métodos usados para las colecciones no son válidos para los mapas.

Constructor de un mapa:

Map unMapa= new {HashTable|TreeMap| HashMap} ();

Principales **métodos**:

- **int size()**: número de nodos del mapa
- **boolean isEmpty()**: indica si el mapa está vacío
- **V put (clave:K , valor:V)** : inserta un nuevo nodo con el par (clave, valor). Si ya existe un nodo con ese valor de la clave, lo sobreescribe con el nuevo nodo y devuelve el valor antiguo.

Machaca el valor antiguo, pero tambien me lo devuelve

- **V get (clave:K)** : recupera el valor del nodo al que corresponde una clave
- **boolean containsKey (clave: K)**: indica si existe un nodo con esa clave de búsqueda

- **boolean containsValue (valor: V):** indica si existe al menos un nodo con ese valor. Es menos eficiente que el método containsKey.
- **Set keySet() :** devuelve una lista de tipo Set con todos las ocurrencias de las claves guardadas en el mapa (como las claves son únicas, deben guardarse en un objeto derivado de Set)
- **Collection values()** : devuelve una lista con las ocurrencias de los valores guardados en el mapa.

Recorrido de un mapa. Se realiza en dos pasos:

1. Obtener una colección **Set con las claves únicas** presentes en el mapa Set

```
misClaves = miMapa.keySet();
```

2. Recorrer la colección de claves con un iterador e ir **obteniendo los valores** que corresponden a cada clave

```
for ( clave K : misClaves){
        // acceder al valor con miMapa.get(clave);
}
```

Hay otra forma de recorrerlos, usando la interfaz Interface Map.Entry<K,V>:

```
// Set<Map.Entry<Integer, String>> listaCompleta = treeMap.entrySet();
// Tengo importado la clase Map.Entry
// El metodo entrySet devuelve todas las parejas del mapa en un conjunto

Set<Entry<Integer, String>> listaCompleta = treeMap.entrySet();
System.out.println(listaCompleta);

// Como es un conjunto, podemos iterar y obtenemos cada objeto de tipo Entry:
// los objetos Entry tienen metodos para recoger clave y valor

Entry<Integer, String> dato;
Iterator<Entry<Integer, String>> itMap = listaCompleta.iterator();
while (itMap.hasNext()) {
        dato = itMap.next();
        System.out.println("Clave: " + dato.getKey() + " -> Valor: " +
        dato.getValue());
}
```

7.- La clase Properties (lo vemos en ficheros)

Es una clase que hereda de HashTable (ver ejemplo) y se puede utilizar para declarar propiedades y parámetros de una aplicación, almacenándolas como pares clave-valor.
Tanto la clave como el valor son normalmente de tipo String, pero las claves no pueden repetirse.

Instanciación:

```
Properties tabla= new Properties();
```

Añadir una propiedad:

```
tabla.setProperty ("clave", "valor");
```

Recuperar una propiedad:

```
Object valor = tabla. getProperty ("clave")
```

Obtener una lista con todos los valores de las claves:

```
Set<Object> claves= tabla.keySet();
```

Devuelve una colección de tipo Set, ya que las claves no pueden repetirse

Recorrer la lista:

```
for (Object key : (Set<Object>) tabla.keySet()){
    tabla.getProperty( (String)key);
}
```

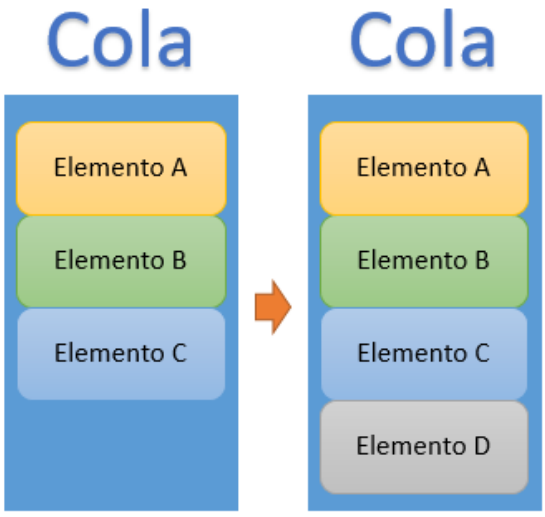
Vaciar la lista:

```
tabla.clear();
```

Además, esta clase dispone de 2 métodos, load(InputStream is) y store(OutputStream os, String mensaje) que permiten, respectivamente, cargar y almacenar los valores de todas las propiedades utilizando un canal de entrada /salida (normalmente un fichero).

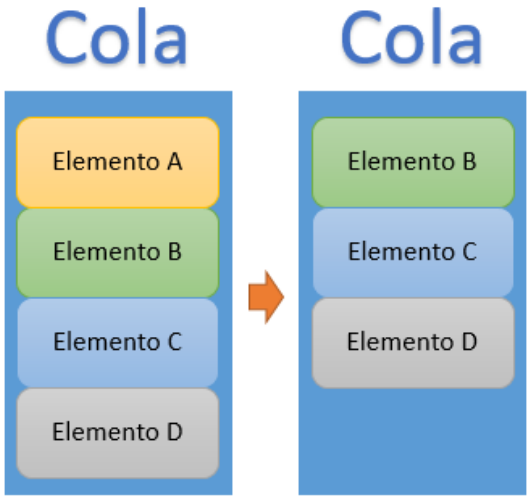
8.- Interfaz Queue<>

Desde el punto de vista teórico una cola es un tipo de dato abstracto (TDA) que sigue el principio FIFO (*first in, first out*) que implica que el primer elemento en ser agregado en la cola es también el primero en ser borrado de la misma.



Agregando

En la imagen se observa una cola con tres elementos, cuando se inserta uno nuevo, este se coloca al final de la cola (elemento D).



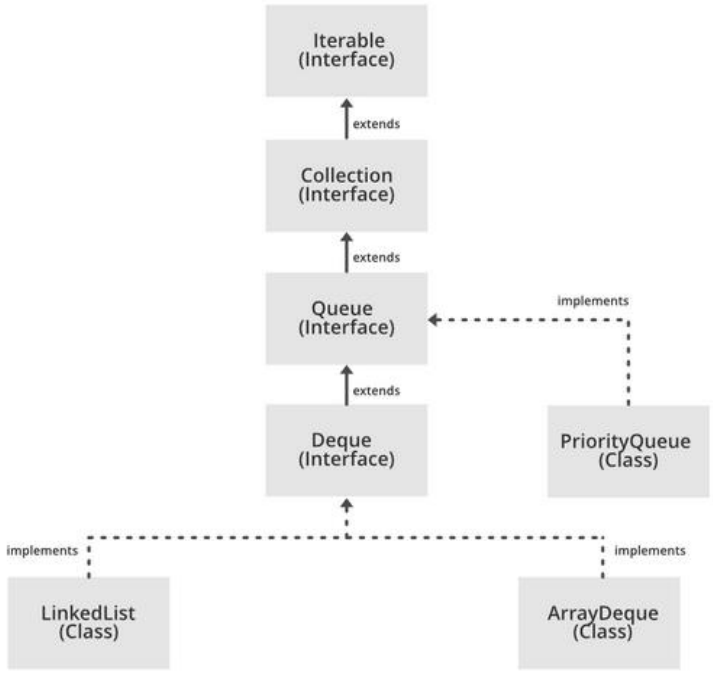
Eliminando

Cuando se elimina un elemento, se quita el primero en haber ingresado, en caso de la cola es el elemento A.

En el mundo real podemos encontrar este ejemplo en las colas de un banco, la cadena de impresión de documentos, etc.

En el caso de la cola en el banco, la primera persona en llegar es también la primera en irse (suponiendo una única ventanilla) y en los documentos a imprimir, la impresora imprime según el orden de llegada.

Colas en Java, distintas opciones



Existen diversas colecciones que permite implementar de forma rápida y sencilla las colas como la interfaz "Queue" o la clase "LinkedList" que ya hemos visto.

Esta situación cambió a partir del agregado a Java de dos interfaces expresamente diseñadas para el manejo de colas:

Queue: Interfaz que extiende a **Collection** y proporciona operaciones para trabajar con una cola.

Deque: Interfaz que representa a una “double-ended queue”, cola doblemente enlazada.

PriorityQueue: Clase para trabajar con colas con prioridades. Implementa la interfaz Queue y ordena los elementos en base a su orden natural, según lo especificado por el método compareTo que hayamos creado en los elementos que implementan **Comparable**, o mediante un objeto **Comparator** que se suministra a través del constructor.

Esta clase proporciona una funcionalidad que permite inserciones en orden y eliminaciones de la parte frontal (desencolar normalmente). Al insertar se hace según una prioridad, de tal forma que el elemento de mayor prioridad se sitúa el primero. Las operaciones más habituales son **offer** para insertar en la posición adecuada según la prioridad, **poll** para desencolar, **peek** para obtener el primero de la cola, **clear** para eliminar todos los elementos de la cola y **size** para obtener el número de elementos de la cola .

Para más información:

https://www.geeksforgeeks.org/linked-list-in-java/

https://www.geeksforgeeks.org/queue-interface-java/

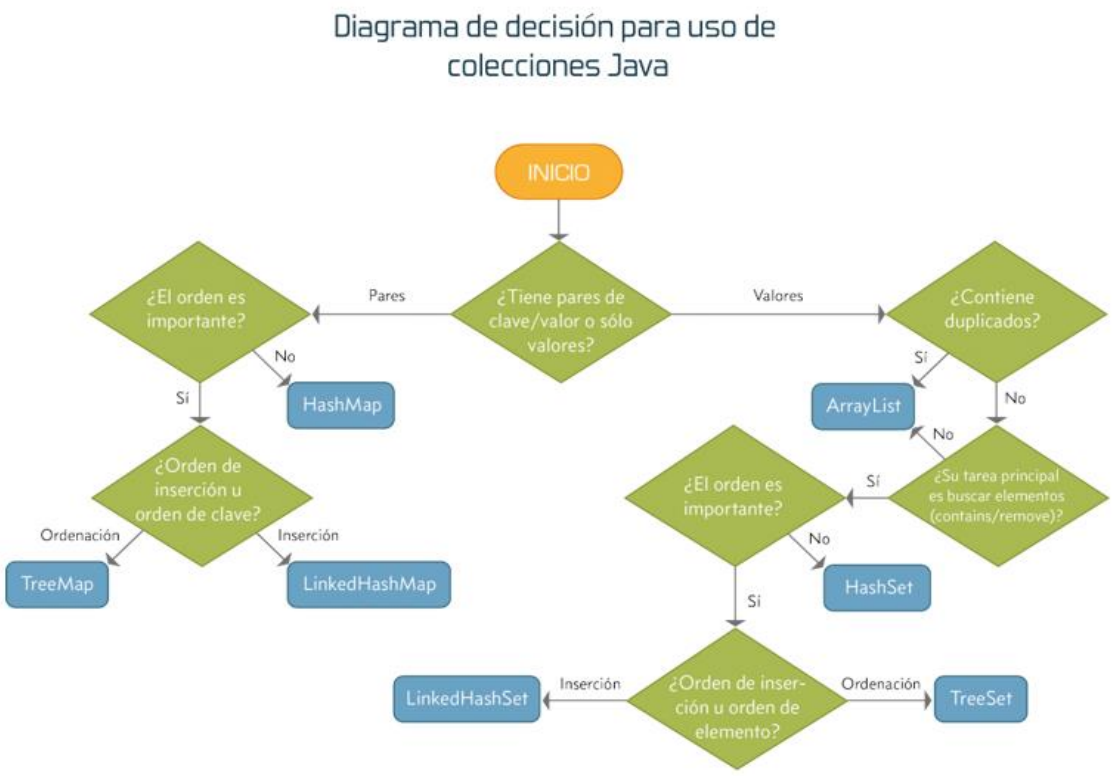
Ejemplos de implementación con Swing:

https://www.tutorialesprogramacionya.com/javaya/detalleconcepto.php?punto=72&codigo=150&inicio=60

9.- Conclusiones

Como hemos visto, Java proporciona una serie de estructuras muy variadas para almacenar datos. Estas estructuras, ofrecen diversas funcionalidades: ordenación de elementos, mejora de rendimiento, rango de operaciones…Un buen uso de estas estructuras mejorará el rendimiento de nuestra aplicación.

Para conocer qué tipo de colección usar, podemos emplear el siguiente diagrama:



Además, gracias a Java 8 y sus *streams*, las operaciones con las colecciones pueden ser mucho más óptimas.