

Programación

Colecciones

Unidad 11

Jesús Alberto Martínez
versión 0.2



Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.
Basado en los apuntes de WirzJava y del CEEDCV



Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

Unidad 11. Colecciones

1 Introducción.....	3
2 Interface List: Listas.....	5
2.1 Clase ArrayList.....	5
2.2 Clase LinkedList.....	6
3 Interface Set: Conjuntos.....	7
3.1 Clase HashSet.....	8
3.2 Clase LinkedHashSet.....	11
3.3 Clase TreeSet.....	11
3.4 Diferencias entre conjuntos.....	12
4 Interface Map: Mapas.....	12
4.1 HashMap.....	13
4.2 TreeMap.....	14
5 Recorrer Colecciones e Iteradores.....	15
5.1 Crear nuestro propio iterador.....	16
5.2 Recorrer mapas.....	18
6 Comparable y Comparator.....	19
6.1 Interfaz Comparable.....	19
6.2 Interfaz Comparator.....	20
7 Últimas Consideraciones.....	22
7.1 Polimorfismo en Colecciones.....	22
7.2 Conversión entre Colecciones.....	23
7.3 Elección de Colección.....	23
7.4 Clases Anónimas.....	24

1 Introducción

Las colecciones representan grupos de objetos, denominados elementos que podemos tratar de una forma conjunta, por ejemplo, recorriéndolos o accediendo a ellos individualmente. Un ejemplo de colección, con la que hemos trabajado a lo largo de este curso son los ArrayList.

Podemos encontrar diversos tipos de colecciones, según si sus elementos tienen una posición determinada o no, o si se permite repetición de elementos o no. En todos los casos, el tamaño de las colecciones es dinámico, esto es, podremos añadir y eliminar los elementos que sea necesario. Para usar estas colecciones haremos uso del Java Collections Framework (JCF), el cual contiene un conjunto de clases e interfaces del paquete java.util para gestionar colecciones de objetos. Una limitación de las colecciones es que solo se aplican a objetos, no a tipos primitivos. Todas las colecciones implementan la interfaz Collection, en la que encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea. Estos métodos generales son:

- boolean **add** (Object o) Añade un elemento (objeto) a la colección. Nos devuelve true si se ha añadido el elemento correctamente, o false en caso contrario.
- void **clear**() Elimina todos los elementos de la colección.
- boolean **contains** (Object o) Indica si la colección contiene el elemento (objeto) indicado.
- boolean **isEmpty**() Indica si la colección está vacía (no tiene ningún elemento).
- Iterator **iterator**() Proporciona un iterador para acceder a los elementos de la colección (lo veremos más en detalle)
- boolean **remove**(Object o) Elimina un elemento (objeto) de la colección, devolviendo true si dicho elemento estaba contenido en la colección, y false en caso contrario.
- int **size**() Nos devuelve el número de elementos que contiene la colección.
- Object [] **toArray**() devuelve la colección como un array de objetos. Para llamarlo, crearemos e instanciaremos previamente el array destino.

```
String [] cadenas = new String[MiColeccion.size()];  
MiColeccion.toArray(cadenas);
```

Esta interfaz es muy genérica, y por lo tanto no hay ningún tipo de datos que la implemente directamente, sino que implementarán subtipos de ellas.

En Java las principales interfaces de las que disponemos para trabajar con colecciones son: Set, List y Map y las clases más interesantes y que vamos a utilizar son las siguientes:

List → ArrayList, Vector y LinkedList

Set → HashSet, TreeSet y LinkedHashSet

Map → HashMap, TreeMap y LinkedHashMap

Queue → Priority Queue, ArrayDeque

A grandes rasgos, las clases que implementan List almacenan los elementos en cierto orden, admiten duplicados y permiten acceder a ellos por su posición. Las clases de Set tienen como principal característica que no admiten duplicados y, por último, los Map permite acceder por claves y valores en vez de por posiciones.

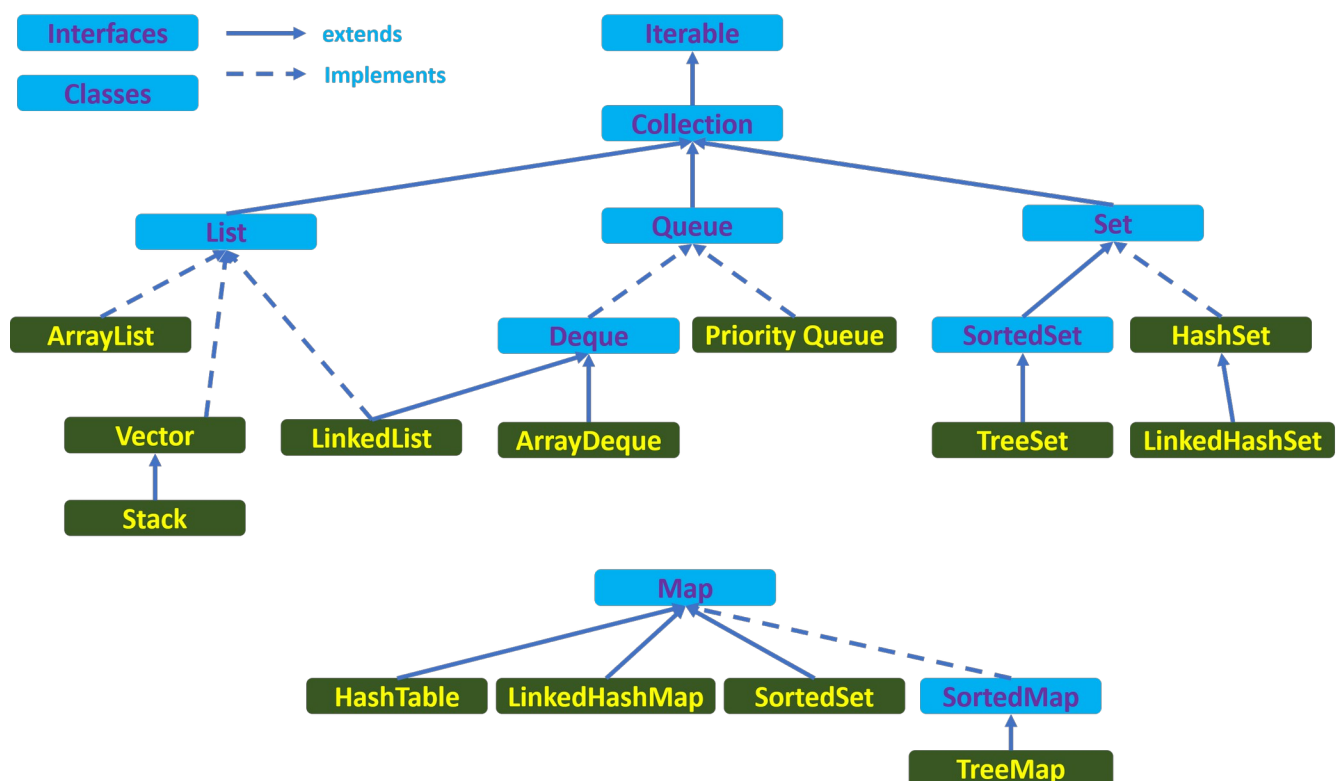


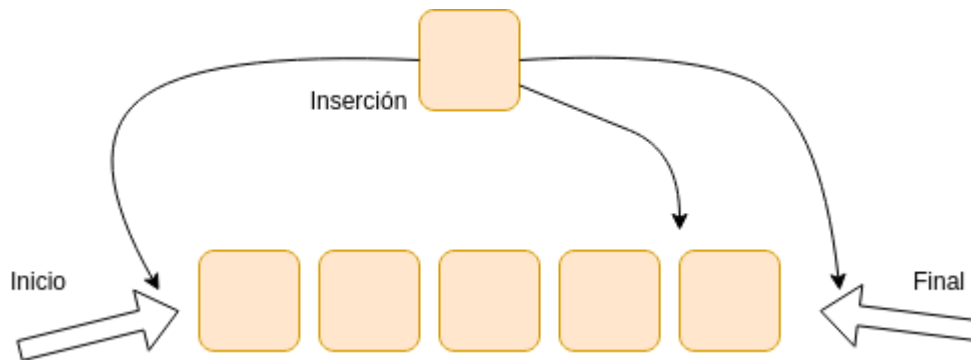
Figura 1: Imagen de <https://chercher.tech/java-programming/collections-java>

Como ya hablamos al usar ArrayList, hay métodos que necesitan comparar objetos para ver si son iguales (por ejemplo: `contains(Object)` compara el elemento pasado con cada elemento de la colección y devuelve `true` si lo encuentra, `remove(Object)` borra el objeto si lo encuentra, etc.). y decíamos que esos métodos requerían redefinir `equals(Object)`. Eso ocurre con el resto de colecciones y, adicionalmente, deberemos redefinir `hashCode()` como explicaremos más adelante.

Sería inabarcable hablar de todas las colecciones, con sus particularidades y métodos, vamos a ver las más importantes, dividiéndolas en las cuatro interfaces principales: List, Set, Queue y Map.

2 Interface List: Listas

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.



Incorpora métodos nuevos, no disponibles en otras colecciones, que hacen referencia a estos índices:

- void **add**(int índice, Object obj): Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.
- Object **get**(int índice): Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.
- int **indexOf**(Object obj): Nos dice cuál es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.
- Object **remove**(int índice): Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.
- Object **set**(int índice, Object obj): Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

Podemos encontrar diferentes clases que implementan esta interfaz: `ArrayList` ya vista en capítulos anteriores, `LinkedList` y las obsoletas `Vector` y su hija `Stack`.

2.1 Clase ArrayList

Esta clase, ya vista en capítulos previos, implementa una lista de elementos mediante un array de tamaño variable. Conforme se añaden elementos el tamaño del array irá creciendo si es necesario. El array tendrá una capacidad inicial, y en el momento en el que se rebase dicha capacidad, se aumentará el tamaño del array.

Las operaciones de añadir un elemento al final del array (add), y de establecer u obtener el elemento en una determinada posición (get/set) tienen un coste temporal constante. Las inserciones y borrados tienen un coste lineal dependiente del número de elementos del array.

2.2 Clase LinkedList

Es como un ArrayList, pero los elementos están conectados con el anterior y el posterior permitiendo gestión tanto por el principio como el final de la lista. Cuando realicemos inserciones, borrados o lecturas en los extremos inicial o final de la lista el tiempo será constante, mientras que para cualquier operación en la que necesitemos localizar un determinado índice dentro de la lista deberemos recorrer la lista de inicio a fin, por lo que el coste será lineal con el tamaño de la lista.

Para aprovechar las ventajas que tenemos en el coste temporal al trabajar con los extremos de la lista, se proporcionan métodos propios para acceder a ellos en tiempo constante:

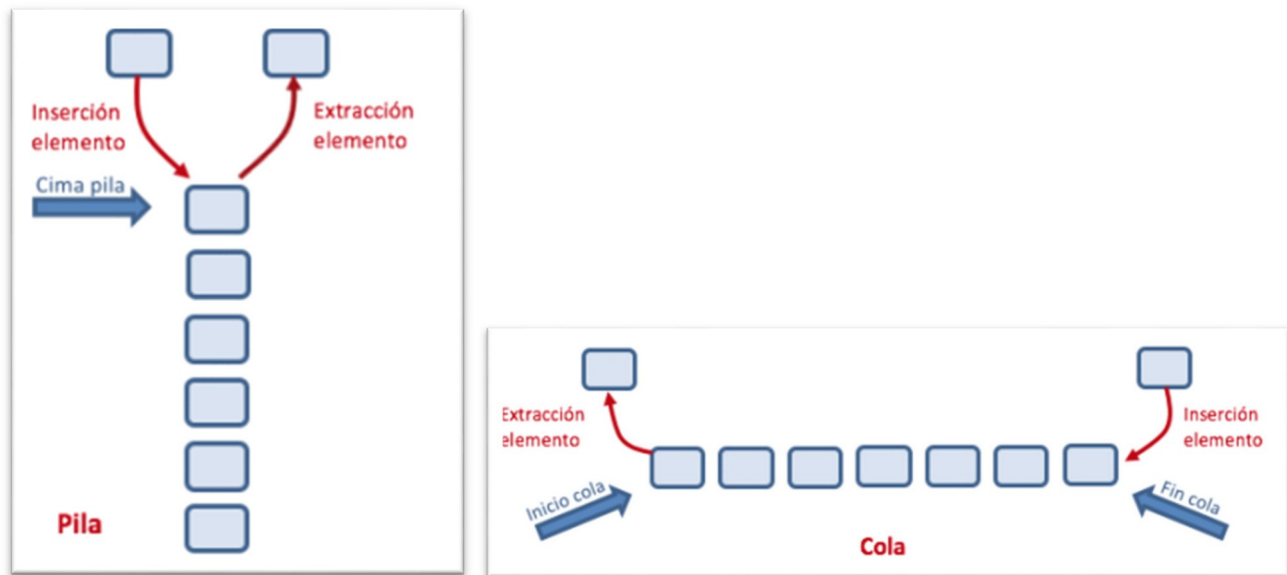
- void **addFirst** (Object o) / void **addLast** (Object o): Añade el objeto indicado al principio / final de la lista respectivamente.
- Object **getFirst**() / Object **getLast**(): Obtiene el primer / último objeto de la lista respectivamente.
- Object **removeFirst**() / Object **removeLast**(): Extrae el primer / último elemento de la lista respectivamente, devolviéndonos dicho objeto y eliminándolo de la lista.

Hemos de destacar que estos métodos nos permitirán trabajar con la lista como si se tratase de una pila o de una cola. En el caso de la pila realizaremos la inserción y la extracción de elementos por el mismo extremo, mientras que para la cola insertaremos por un extremo y extraeremos por el otro.

```
LinkedList <String> lista = new LinkedList<>();
if (lista.size()<100) {lista.addFirst("Pepe");}
String n = lista.removeLast();
if (!parking.isEmpty()) System.out.println (parking.pgetFirst());
```

También implementa la interfaz Queue, con operaciones típicas de colas:

- **poll()** / **pop()**: elimina y devuelve la cabeza de la pila (el primer elemento) Sería similar a removeFirst().
- **push()**:añade un elemento en la cabeza de la pila (el primer elemento). Sería similar a addFirst().
- **peek()**: devuelve la cabeza de la pila - como poll()/pop() - pero sin eliminarlos.



🔊 Las colecciones desde Java9 tienen un método estático de factoría (hace la función de constructor) llamado 'of', y se usa así:

```
List<String> list = List.of("Alpha", "Bravo", "Charlie");
Set<String> set = Set.of("Alpha", "Bravo", "Charlie");
```

📖 Interfaces y clases:

Como ya comentamos en el capítulo de polimorfismo e interfaces, se suelen crear variables o referencias del tipo de la interfaz y se instancian con una clase que implemente la interfaz. La limitación de esta técnica es que los atributos/métodos que podremos usar serán los definidos a la interfaz o nos veremos obligados a hacer castings.

Así pues, será frecuente ver definida colecciones de esta forma:

```
List<String> lista1 = new LinkedList<>();
List<String> lista2 = new ArrayList<>();
```

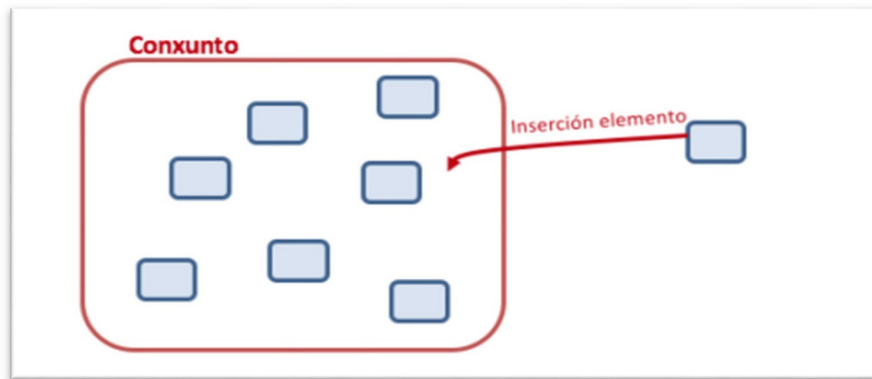
3 Interface Set: Conjuntos

Los conjuntos son grupos de elementos en los que no se permite ningún elemento repetido. Consideramos que un elemento está repetido si tenemos dos objetos o1 y o2 iguales, comparándolos mediante el operador o1.equals(o2).

Implementa los métodos vistos para la interfaz Collection. El método add añadirá el elemento si no está repetido, devolviendo true, y no lo añadirá en caso de ya existir previamente en el conjunto, devolviendo false.

Un conjunto podrá contener a lo sumo un elemento null y podremos recorrerlo con un for-each o mediante un Iterator. Esta última forma la veremos en apartados posteriores.

Esta interfaz es implementada por distintas clases, de las que destacaremos: HashSet, LinkedHashSet y TreeSet.



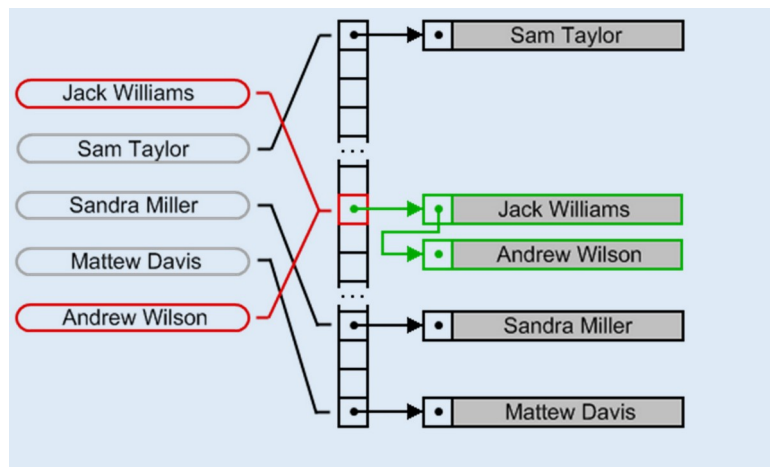
3.1 Clase HashSet

Los objetos de esta clase (como todas las que llevan la partícula hash en su nombre) se almacenan en una tabla de dispersión (hash).

En una estructura hash se almacena cada dato en una posición calculada a partir de una fórmula, una operación sobre sus datos. Así, los datos se dispersan y su acceso es más eficiente que en una estructura no ordenada.

Llevado a la vida real, imagina que tienes que guardar 30 DNI físicamente en una caja. Para acceder a ellos de forma rápida podrías guardarlos ordenadamente pero cada vez que te diesen uno nuevo, insertarlo ordenado sería laborioso (costoso en tiempo). Sabiendo que la letra de los DNI españoles puede tener 23 valores distintos, lo que podrías hacer, es tener 23 cajas, cada una etiquetada con una letra, y meter en cada una los DNI cuya letra sea la de la caja. Los DNI se "dispersarían" por las cajas. La búsqueda sería rápida y añadir nuevos DNI también. En este caso, obtener la letra es la función hash. Si tuvieras 1000 DNI cada caja tendría muchos DNI, lo que te ralentizaría las búsquedas, por lo que te interesaría tener más cajas, necesitarías una función hash distinta. Las "cajas" en Java se llaman "buckets" y que dos DNI vayan a la misma caja se llama "colisión". Lo óptimo es reducir el número de colisiones.

En el siguiente ejemplo, vemos como a partir del nombre se calcula una posición en la estructura, y si hay una colisión (como Jack Williams y Andrew Wilson), se enlazan unos con otros.



Como todas las clases que implementan la interface Set, no admite duplicados. Para identificar los duplicados utilizará los métodos `equals()` y `hashCode()`, que habrá de redefinir como ya comentamos previamente. Recordemos que la definición por defecto de estos métodos en la clase `Object`, compara la referencia de cada objeto, y dos objetos distintos aun con todos los atributos iguales, produciría un `false` en el `equals()` y distinto valor de `hashCode()`.

El método `hashCode()` deberá devolver un valor para cada objeto, de forma que dos objetos que consideremos que son iguales, tengan el mismo `hashCode()`, yendo a la par con `equals()`. Los generadores de código de los IDE nos ayudan en esta labor.



Siempre que redefinamos `equals()` hay que redefinir `hashCode()`, ya que si dos objetos son iguales según `equals()`, sus métodos `hashCode()` deben devolver lo mismo.

Ejemplo de `equals()` y `hashCode()` generado por Netbeans para una clase `Producto`, a partir de su atributo nombre. Las operaciones que se pueden ver en el método buscan la mayor “dispersión” de los elementos.

```

class Producto {
    String nombre;
    double precio;

    Producto (String n, double p){nombre=n; precio=p;}

    @Override
    public int hashCode() {
        int hash = 5;
        hash = 11 * hash + Objects.hashCode(this.nombre);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        final Producto other = (Producto) obj;
        if (!Objects.equals(this.nombre, other.nombre)) return false;
        return true;
    }
}

```

Netbeans usa una forma con números primos para lograr más dispersión, pero podríamos generarlo de forma más sencilla, así:

```
public int hashCode() { return Objects.hash(nombre);}
```

Por último, comentar que esta implementación no trabaja con índices y no garantiza el orden de los elementos a través del tiempo. Como ejemplo podríamos pensar en una lista de la compra ya que no hay elementos repetidos y no nos importa el orden en el cual encontremos los elementos en la lista.

Ejemplo: Lista de la compra (no nos importa el orden, pero no hay repetidos).

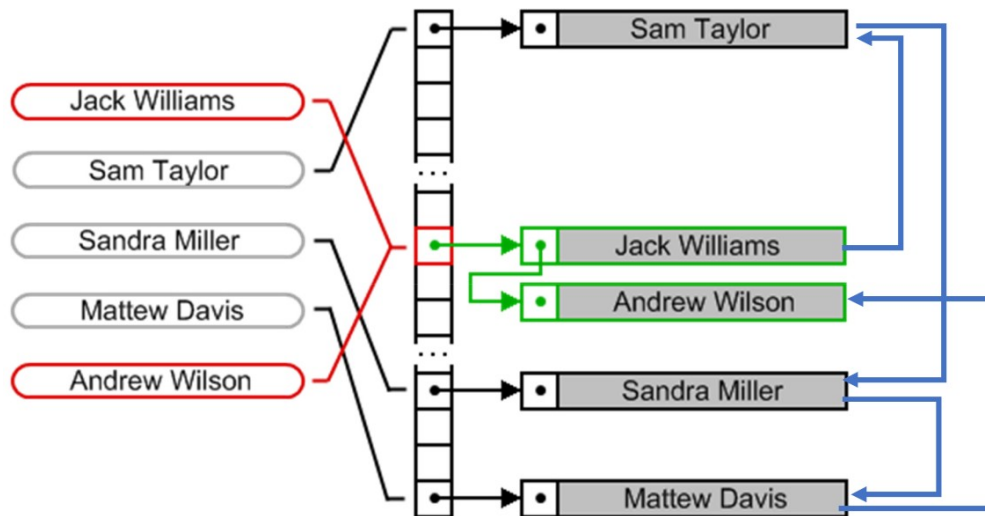
```

HashSet <Producto> listaCompra = new HashSet<>();
if (listaCompra.add(new Producto("Platanos", 2.5)))
    System.out.println("Añadido correctamente");
else System.out.println("No se puede añadir. Repetido");
double total = 0;
for (Producto p: listaCompra)
    total += p.precio;

```

3.2 Clase LinkedHashSet

Es similar al HashSet, pero los elementos, además del almacenamiento de tipo hash, están enlazados entre sí según el orden de inserción, lo que representa una mejora en rendimiento en caso de querer recorrer todo el conjunto, mientras que las operaciones básicas seguirán teniendo coste constante similar a HashSet (salvo la carga adicional que supone tener que gestionar los enlaces).



En el gráfico, las flechas de la derecha representan los enlaces entre los elementos, por orden de inserción.

3.3 Clase TreeSet

Similar a un HashSet, no admite duplicados y está ordenado ascendentemente. Para ambas tareas (duplicados y ordenación) emplea el método **compareTo()** de la clase contenida en el Set o bien el Comparator indicado en el constructor. En un apartado posterior detallaremos estas dos interfaces: Comparable y Comparator y explicaremos ese método en detalle.

A diferencia de TreeSet, HashSet usaba equals() y hashCode() para evitar duplicados.

En secciones posteriores de este capítulo veremos en detalle Comparable y Comparator. Para su almacenamiento usa un árbol en vez de una tabla hash. Por lo tanto, el coste para realizar las operaciones básicas será logarítmico con el número de elementos que tenga el conjunto.

3.4 Diferencias entre conjuntos

Property	HashSet	TreeSet
Ordering or Sorting	HashSet doesn't provide any ordering guarantee.	TreeSet provides ordering /sorting guarantee.
Comparison and Duplicate detection	HashSet uses equals() method for comparison.	TreeSet uses compareTo() method for comparison
Underlying data structure	HashSet is backed by hash table	TreeSet is backed by Red-Black Tree.
Null element	HashSet allows one null element	TreeSet doesn't allows null objects.
Implementation	Internally implemented using HashMap	Internally implemented using TreeMap.
Performance	HashSet is faster	TreeSet is slower for most of the general purpose operation e.g. add, remove and search

¿Cuándo usar cada uno de los tipos de conjuntos?

En todos los casos, usaremos conjuntos cuando no queramos repetidos ni tampoco queramos tener los elementos localizables y accesibles por un índice. Usaremos:

- HashSet: Si no importa el orden y queremos un acceso rápido.
- LinkedHashSet: Si nos hace falta mantener el orden de inserción, penalizando ligeramente la inserción.
- TreeSet: Si nos hace falta mantener el conjunto ordenado por un determinado criterio, penalizando el tiempo de inserción ya que hay que gestionar ese orden.

4 Interface Map: Mapas

Aunque muchas veces se hable de los mapas como una colección, en realidad podemos considerarlas como tal desde un punto conceptual, pero no heredan de la interfaz Collection.

Un mapa es un objeto que relaciona una clave (key) con un valor. Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor. En versiones anteriores este mapeado entre claves y valores lo hacía la clase Dictionary, que ha quedado obsoleta. Tanto la clave como el valor puede ser cualquier objeto.

Un ejemplo típico puede ser una estructura para almacenar País -> Cantidad de Habitantes, en el que el acceso se hace por "país", no por un índice, como tendríamos que hacer con un ArrayList.

Los métodos básicos para trabajar con estos elementos son los siguientes:

- Object **get** (Object clave): Nos devuelve el valor asociado a la clave indicada

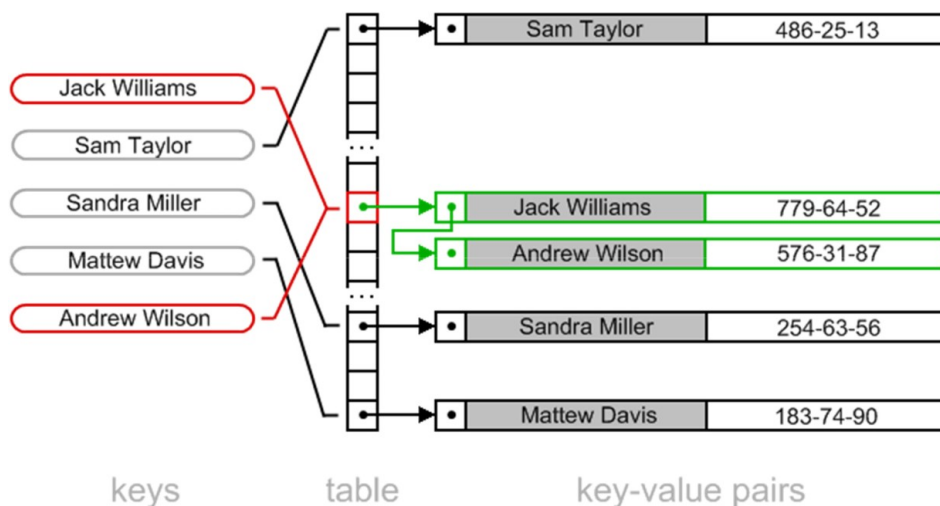
- Object **put** (Object clave, Object valor): Inserta una nueva clave con el valor especificado. Si ya existía esa clave, la reemplaza con el nuevo valor y nos devuelve el valor que tenía antes dicha clave. Si no existía la clave, la añade y devuelve null.
- Object **remove** (Object clave): Elimina una clave, devolviéndonos el valor que tenía dicha clave.
- Set **keySet()**: Nos devuelve el conjunto de claves registradas
- int **size()**: Nos devuelve el número de parejas (clave,valor) registradas.
- Boolean **containsKey** (Object key) devuelve true si la clave key está dada de alta en el Map.

Encontramos distintas implementaciones de los mapas como son HashMap y TreeMap. Cada elemento del mapa será una tupla de tipo Map.Entry, que dispondrá de los métodos getKey () y getValue ().



4.1 HashMap

Utiliza una tabla de dispersión para almacenar la información del mapa con la técnica de hashing que describimos previamente. Las operaciones básicas (get y put) se harán en tiempo constante siempre que se dispersen adecuadamente los elementos. Es coste de la iteración dependerá del número de entradas de la tabla y del número de elementos del mapa.



Características:

- No se garantiza que se respete el orden de las claves y permite una sola clave igual a null y múltiples valores iguales a null.

Existe otra clase llamada 'HashTable' similar a HashMap, que a diferencia de éste no admite ningún valor nulo, ni en la clave ni en el valor y además está sincronizada, lo que la hace aconsejable para aplicaciones multihilo.

- La clave y el valor tienen que ser clases, no tipos primitivos.
- La clase de la clave del mapa tiene que tener definido el método hashCode() ya que a la hora de introducir una nueva clave, es la función que determina si ya existe (y por tanto la sustituirá) o no existe (y la añadirá). Las clases típicas como Integer, String, etc. ya lo tienen definido.

Ejemplo: País con cantidad de habitantes.

```
HashMap < String, Integer > mapaPaises = new HashMap < > ();
mapaPaises.put("España", 47000000);
if (mapaPaises.containsKey("Portugal"))
    mapaPaises.put("Portugal ", mapaPaises.get("Portugal ") + 100000);
else mapaPaises.put("Portugal ", 0);
```

Para recorrerlos podremos utilizar un for...each (o un Iterador como veremos más adelante).

```
for (String k: mapaPaises.keySet())
    System.out.println(k + " tiene " + mapaPaises.get(k) + " habitantes.");
```

4.2 TreeMap

Utiliza un árbol para implementar el mapa de forma que los elementos se encontrarán ordenados por orden ascendente de clave.

Ya que está ordenado incorpora métodos que no tienen sentido para un HashMap pero sí para un mapa ordenado:

firstKey(), firstEntry(), lastKey(), lastEntry() y otros similares.

Ver: https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html#method_summary

```
Entry <String,String> ent = miTreeMap.firstEntry();
System.out.println(ent.getKey()+" ==> "+ent.getValue());
```

Análogamente a lo que ocurre con TreeSet, la clase contenida en la clave del mapa debe implementar Comparable y desarrollar el método compareTo(), en cambio no necesita equals() ni hashCode(). Más adelante detallamos esta interfaz y el método.

El funcionamiento será igual HashMap, con un rendimiento peor en las operaciones básicas que el HashMap en inserción ya que tiene que mantener adicionalmente el árbol de ordenación.

5 Recorrer Colecciones e Iteradores

Hasta ahora hemos recorrido cualquier colección mediante un bucle for...each. En realidad, lo que estamos haciendo es utilizar implícitamente el iterador que tiene definida esa colección.

Un iterador se puede definir como una clase que usa para recorrer estructuras dinámicas como arrays o Colecciones. Los iteradores que usamos en el for...each están definidos como clase internas de la clase sobre la que iteran.

Una **clase interna** es una clase no estática que se define en el cuerpo de otra clase. La gran funcionalidad de estas clases internas es que pueden acceder a los atributos de la clase que las envuelve. Ejemplo:

```
public class Externa {
    // atributos clase externa
    // constructor y métodos
    public class Interna {
        // atributos clase interna
        // constructor y métodos de la clase interna
    } //fin clase interna
}
//fin clase externa
```

Para crear una instancia de la clase interna hay que emplear esta fórmula (suponiendo constructores sin parámetros):

```
ClaseExterna e = new ClaseExterna();
ClaseExterna.ClaseInterna i = e.new ClaseInterna();
```

O bien todo junto:

```
ClaseExterna.ClaseInterna j = new ClaseExterna().new ClaseInterna();
```

Viendo la definición de clase interna del cuadro azul previo, crearemos una instancia del iterador de una colección así:

```
ArrayList<String> lista= new ArrayList<>();
Iterator<String> iter = lista.iterator();
```

Los iteradores siempre proporcionan tres métodos:

- Object item = iter.**next()** para obtener el siguiente elemento de la colección.

- `iter.hasNext()` para saber si quedan más elementos que leer.
- `iter.remove()`; para borrar el último elemento que hayamos leído.

Podremos utilizarlo para recorrer la colección e incluso eliminar aquellos que cumplan ciertas condiciones:

```
while (iter.hasNext()) {  
    String item = iter.next();  
    if (condicion_borrado(item)) iter.remove();  
}
```

Como ya mencionamos estas operaciones las podemos hacer con una sintaxis más sencilla empleando `for...each` (aunque el `for...each` no permite el borrado que sí lo podemos hacer con el método `remove()` del iterador).

Tenemos también el método `descendingIterator()` para instanciar el iterador.

```
Iterator<String> iter = list.descendingIterator();  
while (iter.hasNext()) System.out.println(iter.next());
```

5.1 Crear nuestro propio iterador

En algunos casos nos puede interesar nuestro propio iterador, bien para una colección o bien para una estructura dinámica definida por nosotros. Vamos a distinguir dos situaciones: iteradores externos sobre Colecciones o iteradores internos de clases creadas por nosotros.

Iterador sobre Colección

Las colecciones ya tienen su propio iterador definido (el que usamos con el `for...each`). Si queremos hacer una iteración distinta lo haremos con un iterador de una clase separada. Este sería el código para hacer un recorrido de un `ArrayList` de `Integer`, solo para las posiciones pares: 0,2,4,....

```
class IteradorPosPares implements Iterator<Integer> {  
    private List<Integer> lista;  
    private int posActual = 0;  
    public IteradorPosPares(List<Integer> li) {  
        lista = li;  
    }  
    public boolean hasNext() {  
        return posActual < lista.size();  
    }  
    public Integer next() {
```



```

        Integer result = lista.get(posActual);
        posActual += 2;
        return result;
    }
}

```

para utilizarlo:

```

List<Integer> lista = Arrays.asList(1, 2, 3, 4, 5, 6);
Iterator it = new IteradorPosPares(lista);
while (it.hasNext())
    System.out.println(it.next());

```

Iterador sobre una clase propia:

Para nuestras clases, podemos hacerlo como clase interna, y así hacer que funcione, no solo con la estructura de iterador que acabamos de ver, sino que también con un for...each.

Las condiciones para hacerlo son las siguientes:

- La clase sobre la que queremos iterar debe implementar la interfaz iterable. Las colecciones lo hacen también: `public interface Collection<E> extends Iterable<E> {}`
- La clase sobre la que queremos iterar debe definir el método iterator() de la interfaz Iterable. Ese método Iterator() tiene las siguientes características:
 - Define una clase interna que implementa la interfaz Iterator.
 - Esa clase interna debe escribir los métodos hasNext y next() de la interfaz Iterator.
 - El método devolverá una instancia de esa clase interna.
- Una vez hecho esto, podremos usar un for...each para recorrer los elementos de nuestra clase.

Este es un ejemplo que define una colección llamada MiColección que tiene un array de 20 posiciones, y un iterador para recorrer solo los pares.

```

class MiColeccion implements Iterable<Integer> {
    private final static int TAM = 20;
    private Integer[] arr = new Integer[TAM];
    MiColeccion() {
        for (int i = 0; i<TAM; i++)
        }
    arr[i] = i * 10;
    @Override
    public Iterator<Integer> iterator() {
        Iterator<Integer> it = new MiIterador();
        return it;
    }
}

```

```

    }
    class MiIterador implements Iterator<Integer> {
        private int sig = 0;
        @Override
        public boolean hasNext() {
            if (sig<TAM) return true;
            return false;
        }
        @Override
        public Integer next() {
            int val = arr[sig];
            sig += 2;
            return val;
        }
    }
}


```

Y lo usaríamos así:

```

MiColeccion m = new MiColeccion();
for (Integer i : m)
    System.out.println(i);

```

 La clase MiIterador se puede definir como anónima, pero aún no sabemos cómo hacer eso, lo veremos en capítulos posteriores.

5.2 Recorrer mapas

Vamos a ver las 2 formas de recorrer un HashMap, usando entrySet() o usando keySet().

Con keySet() lo que se obtiene como indica el nombre de la función son las claves y mediante un iterador se recorre la lista de claves. De esta forma si queremos saber también el valor de cada elemento tenemos que usar la función get(clave).

```

HashMap<String,Float> listaProductos;
Iterator<String> productos = listaProductos.keySet().iterator();
while(productos.hasNext()){
    String clave = productos.next();
    System.out.println(clave + " - " + listaProductos.get(clave));
}

```

La otra opción es entrySet() con la que se obtienen los valores y al igual que en el caso anterior con un iterador se recorre el HashMap, pero de esta forma hay que crear una variable de tipo Map.Entry para almacenar el elemento y con los métodos getKey() y getValue() de Map.Entry se obtienen los valores.

Se puede usar un iterador del tipo que vamos a coger en este caso Map.Entry de la misma forma que se hizo en el método anterior, o sino usar un iterador genérico y luego hacer el casting a Map.Entry.

```
HashMap<String, Float> listaProductos;  
Iterator iterador = listaProductos.entrySet().iterator();  
Map.Entry producto;  
while (iterador.hasNext()) {  
    producto = (Map.Entry) iterador.next();  
    System.out.println(producto.getKey() + " - " + producto.getValue());  
}
```

Con esta segunda forma es necesario usar una variable Map.Entry. El resultado es el mismo, aunque esta segunda forma es más eficiente puesto que mientras que en la primera solo obtenemos la clave y luego hay que buscar el contenido asociado con la función get(clave) con esta segunda forma ya tenemos ambos valores y no hay que realizar esa búsqueda adicional, aunque quizás la primera forma sea más sencilla.

6 Comparable y Comparator

6.1 Interfaz Comparable

Comparable es una interfaz que nos permite ordenar los elementos de una colección según los criterios que queramos. Esta interfaz debe ser implementada por la clase que forma la colección. La forma de ordenar (que atributos son los que definen el orden, orden ascendente o descendente, etc.) la marca el método abstracto compareTo<T>(T objeto).

La clase debe redefinir este método de forma que devuelva un entero negativo si el objeto es menor que el pasado como parámetro, un entero positivo si el objeto es mayor que el pasado como parámetro, 0 si son iguales. Y ese será el criterio de ordenación.

Este método es usado por TreeSet y TreeMap (implementando Comparable) para mantener su orden interno y evitar duplicados, y también será el que se emplee para ordenar listas cuando hacemos Collection.sort (milista);

Este ejemplo ordena una lista de películas ascendentemente por su año.

```
public class Main {  
    public static void main(String[] args) {  
        List<Peli> lista = new ArrayList<>();  
        lista.add(new Peli("Episode 7: The Force Awakens", 2015));  
        lista.add(new Peli("Episode 4: A New Hope", 1977));  
    }  
}
```

```

        lista.add(new Peli("Episode 1: The Phantom Menace", 1999));
        Collections.sort(lista);
        for (Peli p: lista) System.out.println(p);
    }
}

class Peli implements Comparable<Peli> {
    public String nombre;
    public int año;
    @Override
    public int compareTo(Peli p) {
        return this.año - p.año;
    }
    public Peli(String n, int a) {
        this.nombre = n;
        this.año = a;
    }
    @Override
    public String toString() {
        return "(" + this.año + ") " + this.nombre;
    }
}

```

⚡ ¡Ojo! Un TreeSet de Peli estaría ordenado también por año, pero por otra parte no habría duplicados por año...algo un poco extraño, así que compareTo() debería ser análogo a equals() y hashCode() y por ejemplo, en este caso, trabajar sobre 'nombre' y no 'año'.

El problema de la interfaz comparable es si queremos ordenar la colección por distintos criterios, es decir imaginemos que queremos que la lista de películas en algún momento esté ordenada por el año, pero en otros casos, por el nombre. La solución la proporciona la interfaz Comparator.

6.2 Interfaz Comparator

Aunque la misión de Comparator es igual a la de Comparable, lo hace mediante una clase externa, no implementada mediante la clase objeto de ordenación. Por lo tanto, podemos crear tantas clases que la implementen, cada una con sus criterios.

Esa es la gran diferencia: con Comparable solo podemos tener un criterio de ordenación para una clase (criterio definido en la propia clase), y con Comparator podemos ordenar por distintos criterios, (criterios definidos en clases independientes) y usaremos el que queramos en cada ocasión.

Los pasos a seguir serán:

1. Crear la clases que implementan Comparator y redefinen compare () con los mismos criterios que expusimos previamente para compareTo().

2. Crear instancias de las clases creadas en el paso anterior.
3. Llamar a `Collection.sort()` incluyendo como segundo parámetro la instancia que queramos que marque la ordenación.

Siguiendo el ejemplo anterior, si queremos mostrar las películas primero ordenadas por año y luego por nombre, haríamos lo siguiente:

```
public class Main {
    public static void main(String[] args) {
        List < Peli > lista = new ArrayList < > ();
        lista.add(new Peli("Episode 7: The Force Awakens", 2015));
        lista.add(new Peli("Episode 4: A New Hope", 1977));
        lista.add(new Peli("Episode 1: The Phantom Menace", 1999));

        ComparaNombre compNombre = new ComparaNombre();
        Collections.sort(lista, compNombre);
        for (Peli p: lista) System.out.println("Por nombre: " + p);

        ComparaAño compAño = new ComparaAño();
        Collections.sort(lista, compAño);
        for (Peli p: lista) System.out.println("Por año: " + p);
    }
}

class Peli {
    public String nombre;
    public int año;
    public Peli(String n, int a) {
        this.nombre = n;
        this.año = a;
    }
    @Override
    public String toString() {
        return "(" + this.año + ") " + this.nombre;
    }
}

class ComparaAño implements Comparator<Peli> {
    public int compare(Peli p1, Peli p2) {
        return p1.año - p2.año;
    }
}

class ComparaNombre implements Comparator {
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1;
        Peli p2 = (Peli) o2;
        return p1.nombre.compareToIgnoreCase(p2.nombre);
    }
}
```

```
}
```

✈ (*) `compareToIgnoreCase()` es un método de `String` que compara la instancia que invoca el método con la cadena pasada como parámetro sin tener en cuenta mayúsculas o minúsculas. Devuelve negativo si la instancia es menor alfabéticamente que el parámetro, 0 si iguales, o positivo si la instancia es mayor, o sea, que encaja perfectamente con lo que queremos hacer en el método `compare` de `Comparator`.

Fíjate que para ordenar por `Comparable` hay que modificar la clase que queremos que sufra la ordenación: hay que decir que implementa `Comparable` y hay que sobrescribir el método `compareTo()`. Esto no ocurre con `Comparator`, no hace falta modificar la clase, está todo en las clases que implementan `Comparator`. Este puede ser otro motivo para emplear `Comparator` en vez de `Comparable`, si no podemos o no queremos modificar la clase que va a sufrir la ordenación.

Se podrá ver la clase de que implementa `Comparator` de forma anónima, sin crear la clase como tal, escribiendo su código como parámetro de `Collections.sort()`, con una sintaxis más compacta:

```
Collections.sort(lista, new Comparator<Peli>(){
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año-p2.año;
    }
});
```

Explicaremos esta estructura más adelante, en el apartado de “Clases anónimas”.

7 Últimas Consideraciones

7.1 Polimorfismo en Colecciones

Como vimos en temas anteriores, podemos crear instanciar una clase sobre una variable de una superclase. Lo mismo ocurre con las interfaces y por lo tanto con las colecciones. Por ejemplo, todas las clases de listas implementan la interfaz `List`. Por lo tanto, en un método que acepte como parámetro un objeto de tipo `List` podremos utilizar cualquier tipo que implemente esta interfaz, independientemente del tipo concreto del que se trate.

Es por lo tanto recomendable hacer referencia siempre a estos objetos mediante la interfaz que implementa, y no por su clase concreta. De esta forma posteriormente podríamos cambiar la

implementación del tipo de datos sin que afecte al resto del programa. Lo único que tendremos que cambiar es el momento en el que se instancia.

```
List<Peli> lista = new ArrayList<>();
//List<Peli> lista = new LinkedList<>();
lista.add(new Peli ("Episode 4: A New Hope", 1977));
```

7.2 Conversión entre Colecciones

Para hacer conversiones entre uno tipos de colecciones y otros siempre podemos hacerlo “a mano”, esto es, un bucle for...each o un iterador e insertar cada elemento de la colección origen en la destino pero existen ciertas conversiones que podemos hacer de forma más sencilla.

- Convertir un Mapa en dos ArrayList, uno para claves y otro para los valores:

```
HashMap<String, Integer> mapaPaíses = new HashMap<>();
ArrayList<String> keyList = new ArrayList<String>( mapaPaíses.keySet());
ArrayList<Integer> valueList = new ArrayList<Integer>( mapaPaíses.values());
```

- El proceso inverso lo podremos hacer también, pero con Streams y una función Lambda. En el tema 15 se tratan los Streams y las funciones Lambda.
- Crear una lista (interfaz List) a partir de un array con:

```
List<String> lista = Arrays.asList(array);
```

- Crear un conjunto desde una lista

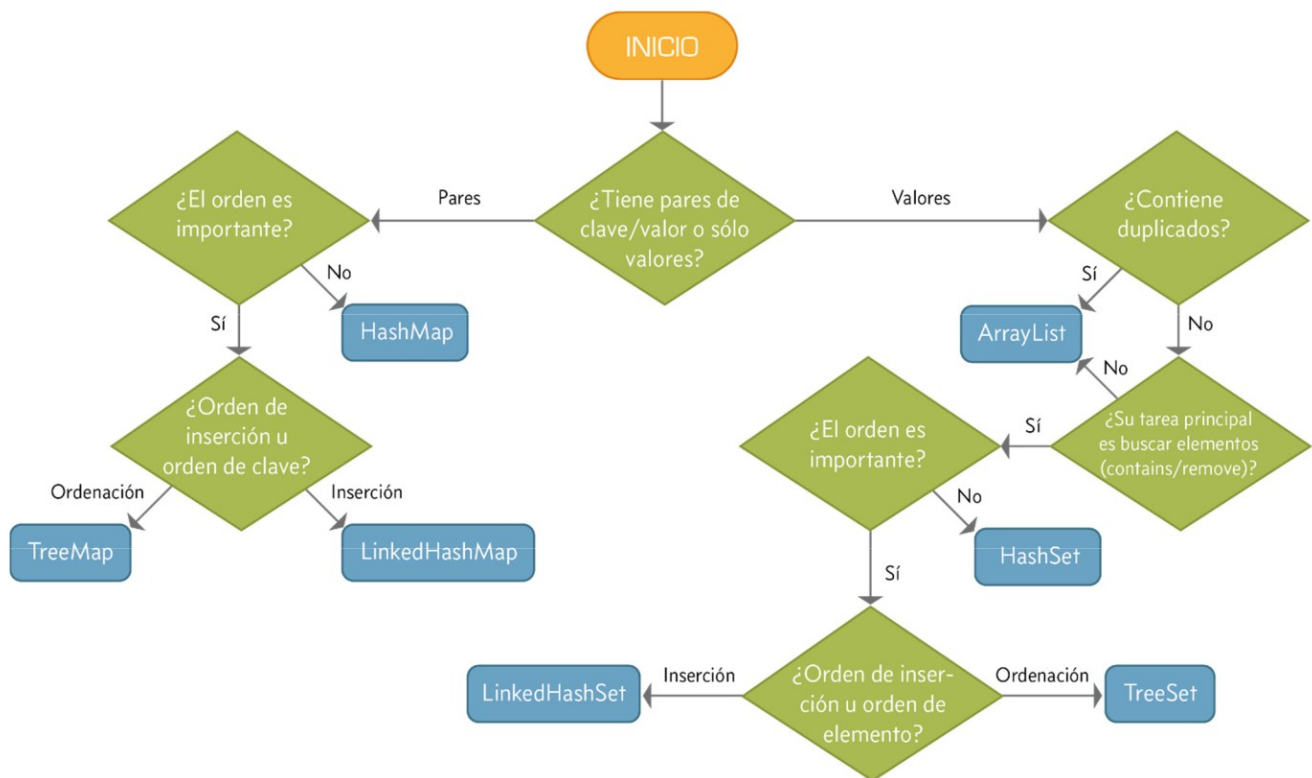
```
Set conjunto = new HashSet (lista);
```

Esto es una forma rápida de ver si una lista contiene duplicados. Hacemos un conjunto con una instrucción como la anterior y como un Set no puede tener duplicados, la comparación de size() de las dos colecciones nos dirá si hay duplicados o no.

7.3 Elección de Colección

A la hora de elegir la colección adecuada para resolver un problema emplearemos diferentes criterios. El primero será la estructura necesaria: admite duplicados o no, queremos que tengan un índice, está ordenado, es de tipo par clave/valor, etc.

El siguiente gráfico puede ser de ayuda para la elección



Con respecto al rendimiento en las distintas operaciones, este podría ser un resumen:

- Las operaciones mediante hash son muy eficientes en operaciones básicas (insertar, borrar y buscar) sobre todo si la función de dispersión es adecuada. La iteración a través de sus elementos es más costosa ya que no mantiene ningún enlace entre los elementos.
- Las colecciones que incorporan link penalizan un poco las operaciones básicas ya que tienen que gestionar los enlaces entre los elementos para mantener el orden de inserción, pero mejoran las iteraciones respecto al hash.
- Los Tree incorporan un sistema de enlaces más complejo y solo son adecuadas cuando necesitamos ordenación.

7.4 Clases Anónimas

Permiten definir e instanciar la clase a la vez, sin asignarle un nombre y para ser usadas una sola vez. Podemos crearlas en el cuerpo de un método, de una clase o como argumento de un método.

El constructor new puede hacer referencia a una interfaz, y entenderemos que está creando una clase que la implementa, o bien hacer referencia a una clase, y entenderemos que está definiendo una clase hija. En ambos casos crea una instancia, pero la clase creada no tiene nombre. Tampoco tendrá nunca constructor, solo llama implícitamente a super() pero no tiene constructor propio.

Por ejemplo

```
Collections.sort(lista, new Comparator<Peli>(){
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año-p2.año;
    }
});
```

`new Comparator<Peli>` está creando una instancia de una clase (sin nombre!!) que implementa la interface `Comparator` (genérica). Luego abre llaves y ahí hace la implementación de la clase, en este caso solo con el método `compare()`.