Interface Collection

Es una interface de Java, por tanto no se puede construir, es decir no puedo crearme un objeto de tipo Colletion (new). Todas las clases que implementen una Collection si que nos van a permitir utilizar sus operaciones

Entre sus operaciones ellas están:

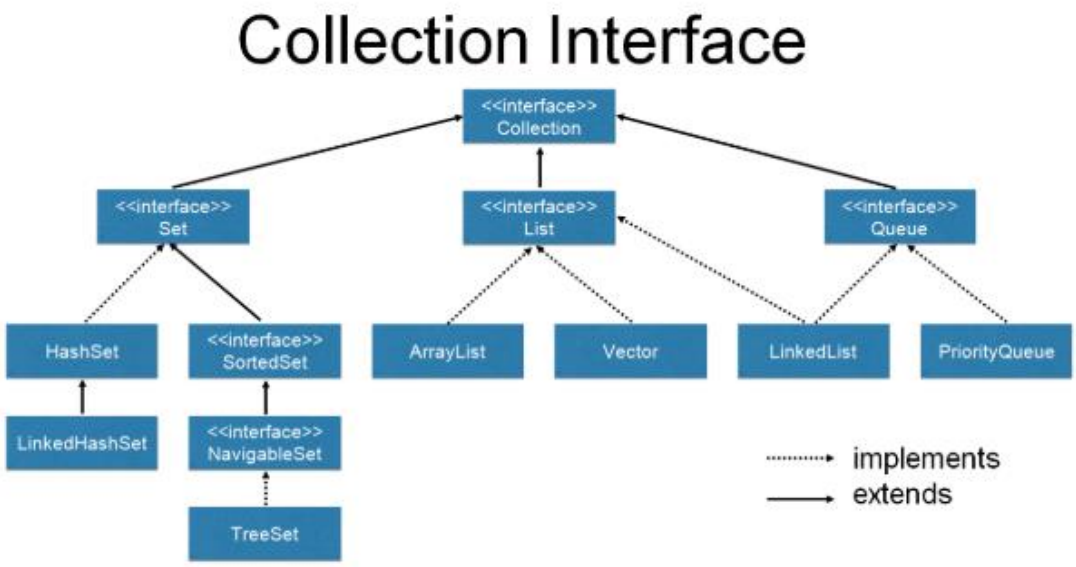
- ✓ **add(e):** nos permite añadir un elemento a la colección
- ✓ **size():** devuelve el número de elementos almacenados en la colección.
- ✓ **iterator():** permite recorrer los elementos de la colección.
- ✓ **containts(e):** devuelve si el elemento esta en la colección.
- ✓ **Remove().** Podemos eliminar un elemento dentro de la colección, mientras lo estamos recorriendo.

Hemos de tener en cuenta, que los elementos de una colección:

- Pueden estar repetidos
- Su orden no es relevante (si recorro la colección, el orden de los elementos de una a otra puede cambiar)

Hay tres interfaces que extienden de **Colletion**:

- ➢ **Set**
- ➢ **List**
- ➢ **Queue**



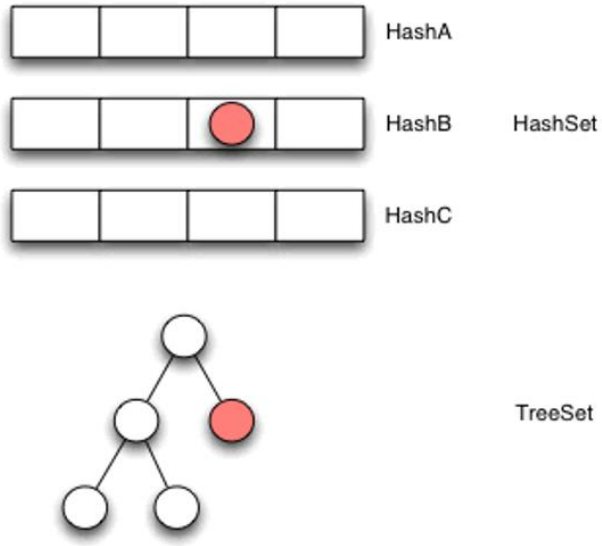
Set

La interfaz **Set** define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos equals y hashCode. Para

comprobar si dos Set son iguales, se comprobarán si todos los elementos que los componen son iguales sin importar en el orden que ocupen dichos elementos.

Dentro de la interfaz Set existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **HashSet:** este implementación almacena los elementos en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Es la implementación más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeSet:** esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento que HashSet. Los elementos almacenados deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashSet:** esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más costosa que HashSet.



List

La interfaz **List** define una sucesión de elementos. A diferencia de la interfaz Set, la interfaz List sí admite elementos duplicados. A parte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

- ***Acceso posicional a elementos:*** manipula elementos en función de su posición en la lista.
- ***Búsqueda de elementos:*** busca un elemento concreto de la lista y devuelve su posición.
- ***Iteración sobre elementos:*** mejora el Iterator por defecto.

- ***Rango de operación:*** permite realizar ciertas operaciones sobre ragos de elementos dentro de la propia lista.

Dentro de la interfaz **List** existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **ArrayList:** esta es la implementación típica. Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.

- **LinkedList:** esta implementación permite que mejore el rendimiento en ciertas ocasiones. Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

ArrayList

Queue

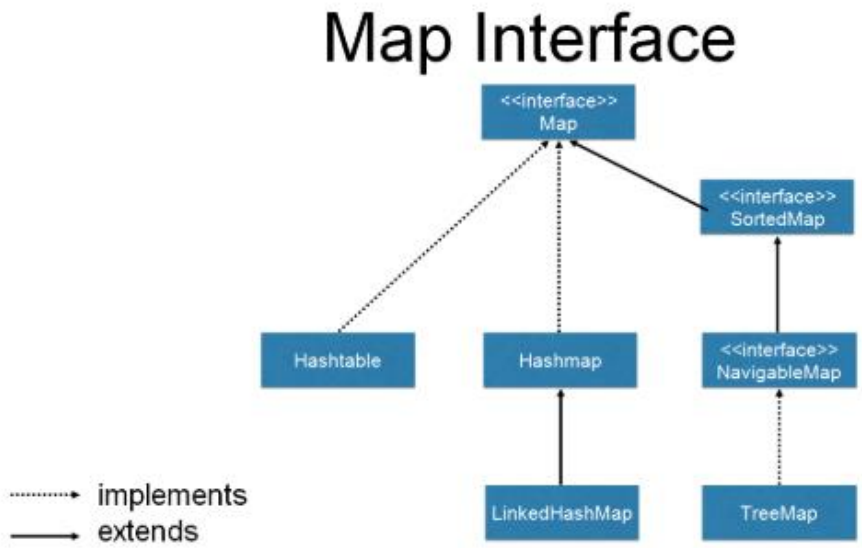
Se conoce como cola a una colección especialmente diseñada para ser usada como almacenamiento temporal de objetos a procesar. Las operaciones que suelen admitir las colas son “encolar”, “obtener siguiente”, etc. Por lo general las colas siguen un patrón que en computación se conoce como FIFO (por la sigla en inglés de “First In - First Out” - “lo que entra primero, sale primero”), lo que no quiere decir otra cosa que lo obvio: El orden en que se van obteniendo los “siguientes” objetos coincide con el orden en que fueron introducidos en la cola. Esto análogo a su tocaya del supermercado: La gente que hace la cola va siendo atendida en el orden en que llegó a ésta.

Hasta hace poco, para implementar una cola FIFO en Java la única opción provista por la biblioteca de colecciones era LinkedList. Sin embargo, aunque la implentación es la correcta, a nivel de interfaz dejaba algo que desear. Los métodos necesarios para usar una LinkedList como una cola eran parte solamente de la clase LinkedList, no existía ninguna interfaz que abstraiga el concepto de “cola”. Esto hacía imposible crear código genérico que use indistintamente diferentes implementaciones de colas (que por ejemplo no sean FIFO sino que tengan algún mecanismo de prioridades). Esta situación cambió recientemente a partir del agregado a Java de dos interfaces expresamente diseñadas para el manejo de colas: La interfaz Queue tiene las operaciones que se esperan en una cola. También se creó Deque, que representa a una “double-ended queue”, es decir, una cola en la que los elementos pueden añadirse no solo al final, sino también “empujarse” al principio.

Map

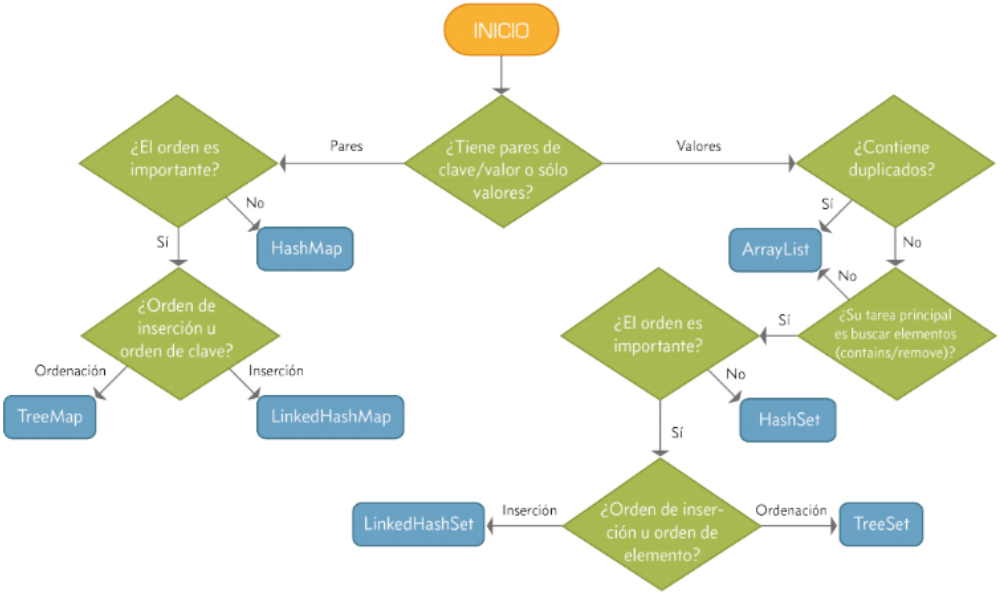
Hemos de decir que un Map no extiende de Colletion pero si es un tipo de Colección.

La interfaz **Map** asocia claves a valores. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.



Dentro de la interfaz Map existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- **HashMap**: este implementación almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- **TreeMap**: esta implementación almacena las claves ordenándolas en función de sus valores. Es bastante más lento que HashMap. Las claves almacenadas deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- **LinkedHashMap**: esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que HashMap.


INICIO
¿Tiene pares de clave/valor o sólo valores?
Pares
Valores
¿El orden es importante?
No
HashMap
Sí
¿Orden de inserción u orden de clave?
Ordenación
TreeMap
Inserción
LinkedHashMap
¿Contiene duplicados?
Sí
ArrayList
No
¿Su tarea principal es buscar elementos (contains/remove)?
No
Sí
¿El orden es importante?
No
HashSet
Sí
¿Orden de inserción u orden de elemento?
Inserción
LinkedHashSet
Ordenación
TreeSet