Collections

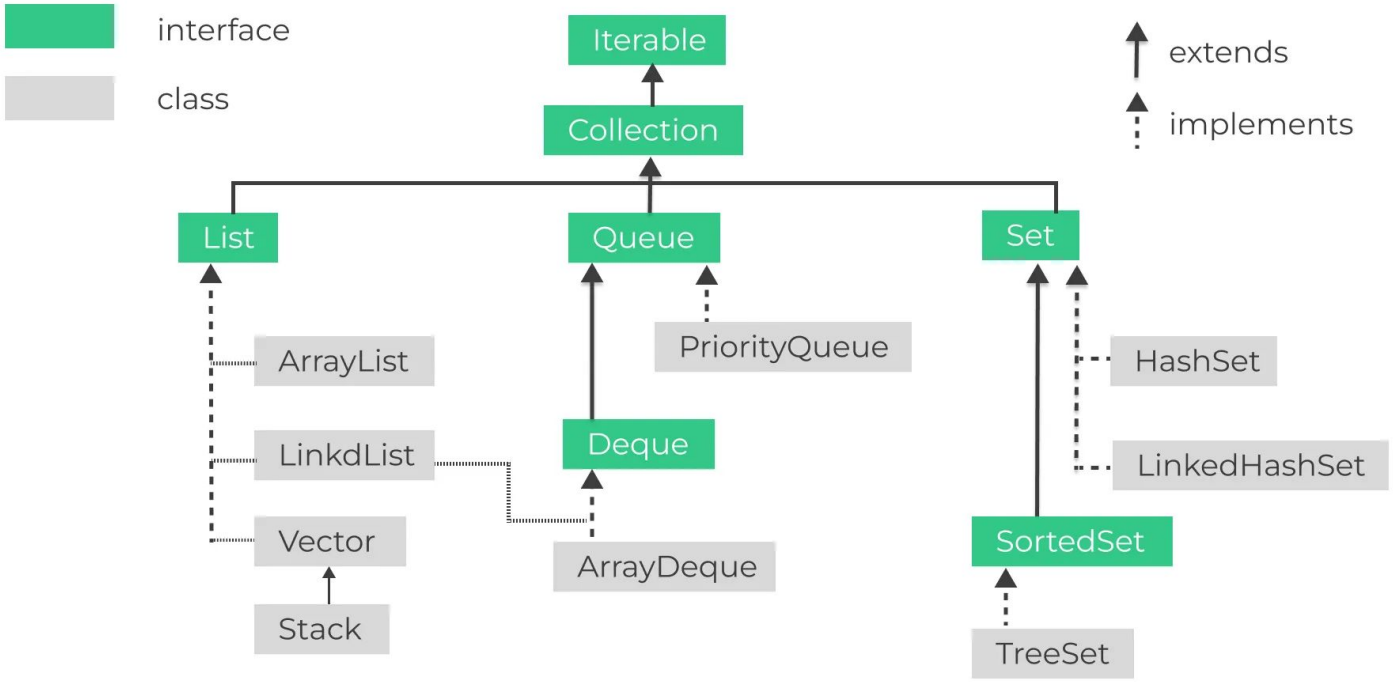
Framework Collection, distintos tipos e implementaciones de cada uno

Collections

En Java, **"Collections" es un marco de trabajo (framework)** que proporciona una arquitectura unificada para trabajar con colecciones de objetos. Una colección es simplemente un grupo de objetos, y el marco Collections de Java proporciona una serie de interfaces y clases para trabajar con diferentes tipos de colecciones de manera eficiente y fácil. Las colecciones pueden contener objetos de cualquier tipo, incluidos los tipos primitivos, y pueden ser de tamaño fijo o variable.

Collection, como interfaz, es “la raíz” de todas las interfaces y clases relacionadas con colecciones de elementos. Algunas colecciones pueden admitir duplicados de elementos dentro de ellas, mientras que otras no admiten duplicados. Otras colecciones pueden tener los elementos ordenados, mientras que en otras no existe orden definido entre sus elementos. Java no define ninguna implementación de esta interface y son respectivamente sus subinterfaces o subclases las que implementarán sus métodos.

Hierarchy of the Collection Framework



List

List en Java es una subinterfaz de la interfaz Collection que define una **colección ordenada de elementos**, es decir, una secuencia de elementos que se pueden acceder por su posición en la lista. Las clases que implementan esta interfaz son:

ArrayList: Implementa una lista respaldada por un arreglo dinámico. Es muy eficiente para el acceso aleatorio y la inserción en la última posición de la lista, pero no es eficiente para la inserción o eliminación de elementos en posiciones intermedias de la lista, ya que los elementos tienen que ser desplazados en el arreglo.

LinkedList: Implementa una lista doblemente enlazada, donde cada elemento de la lista mantiene una referencia al siguiente y al anterior elemento. Es eficiente para la inserción y eliminación de elementos en cualquier posición de la lista, pero menos eficiente para el acceso aleatorio, ya que los elementos tienen que ser recorridos secuencialmente.

Vector: Implementa una lista respaldada por un arreglo dinámico, similar a ArrayList, pero con métodos sincronizados, lo que lo hace seguro para usar en entornos multihilo, es decir, se puede compartir una misma instancia entre varios hilos. Sin embargo, debido a los bloqueos sincronizados, su rendimiento es generalmente inferior al de ArrayList.

Stack: Es una estructura de tipo LIFO (last in - first out), es decir, una pila, donde el último que ingresa es el primero que sale.. Es una subclase de Vector y proporciona los métodos push() y pop() para agregar y eliminar elementos de la pila.

Cada clase que implementa la interfaz List en Java proporciona una implementación diferente de la lista, con diferentes ventajas y desventajas en términos de eficiencia, seguridad y rendimiento. Es importante elegir la clase de lista adecuada para cada situación, según los requisitos específicos de la aplicación.

Todas las clases que implementen List tienen una serie de métodos compartidos además de los propios, ahorrándonos trabajo para memorizar los principales comportamientos de estas estructuras.

Métodos de la interfaz List

A continuación podrán ver algunos de los métodos que comparten las clases que implementan List:

- **add(E e):** agrega un elemento al final de la lista. Indica true o false si se pudo agregar.
- **add(int index, E element):** inserta un elemento en una posición específica de la lista.
- **remove(int index):** elimina el elemento en una posición específica de la lista.
- **get(int index):** obtiene el elemento en una posición específica de la lista.
- **set(int index, E element):** reemplaza el elemento en una posición específica de la lista con un nuevo elemento.
- **indexOf(Object o):** devuelve la posición del primer elemento igual al objeto especificado en la lista.
- **lastIndexOf(Object o):** devuelve la posición del último elemento igual al objeto especificado en la lista.
- **subList(int fromIndex, int toIndex):** devuelve una vista de sublista de la lista original que va desde la posición fromIndex hasta la posición toIndex.

Queue

La interfaz **Queue** en Java es una colección que representa una cola de elementos. La cola es una estructura de datos que sigue el principio de **"primero en entrar, primero en salir"** (FIFO, por sus siglas en inglés). La interfaz Queue es implementada por varias clases e interfaces, a continuación se describen brevemente algunas de ellas:

LinkedList: Igual que en List, ya que esta clase implementa ambas interfaces.

PriorityQueue: Implementa una cola de prioridad donde cada elemento se ordena en función de su prioridad y no permite nulos. Los elementos se agregan en orden de prioridad y se eliminan de la cola en orden de la prioridad más alta, por ende, no están en ningún orden específico como en List. La prioridad que se tiene en cuenta es la dada en el método compareTo() del objeto, siendo el mayor como el “mejor” teniendo más prioridad. Si los elementos no implementan Comparable, entonces se lanzará una excepción en tiempo de ejecución.

ArrayDeque: Implementa una cola de doble extremo, donde los elementos se pueden agregar y eliminar desde ambos extremos de la cola. Es muy eficiente para la inserción y eliminación de elementos en los extremos de la cola, pero menos eficiente para la inserción y eliminación de elementos en el medio de la cola.

Los métodos utilizados para una colección de tipo Deque son:

- **poll():** Remueve pero no devuelve el elemento que está en la cabecera de la cola.
- **pek():** Devuelve el primer elemento de la cola sin eliminarlo.
- **addFirst(E e)** o **addLast(E e):** Inserta un elemento en la cabecera o al final de la cola.
- **removeLast()** o **removeFirst()**: Remueve el último o el primer elemento de la cola. Devuelve true o false indicando si se pudo remover.

Se puede utilizar peek() y poll() tanto por el inicio como por el final de la cola. Es decir, se puede utilizar **peekFirst()** o **peekLast()** para obtener el primer o el último elemento de la cola, respectivamente. Lo mismo aplica para el método poll().

Set

La interfaz **Set** en Java es una colección que **no permite elementos duplicados**. Es decir, cada elemento en un Set debe ser único. Las clases e interfaces que implementan a Set son:

HashSet: Esta clase implementa la interfaz Set utilizando una tabla hash para almacenar los elementos. Los elementos no se mantienen en orden específico. Permite nulos.

LinkedHashSet: Al igual que HashSet utiliza una tabla hash, sin embargo, los elementos se mantienen en orden de inserción. También permite nulos.

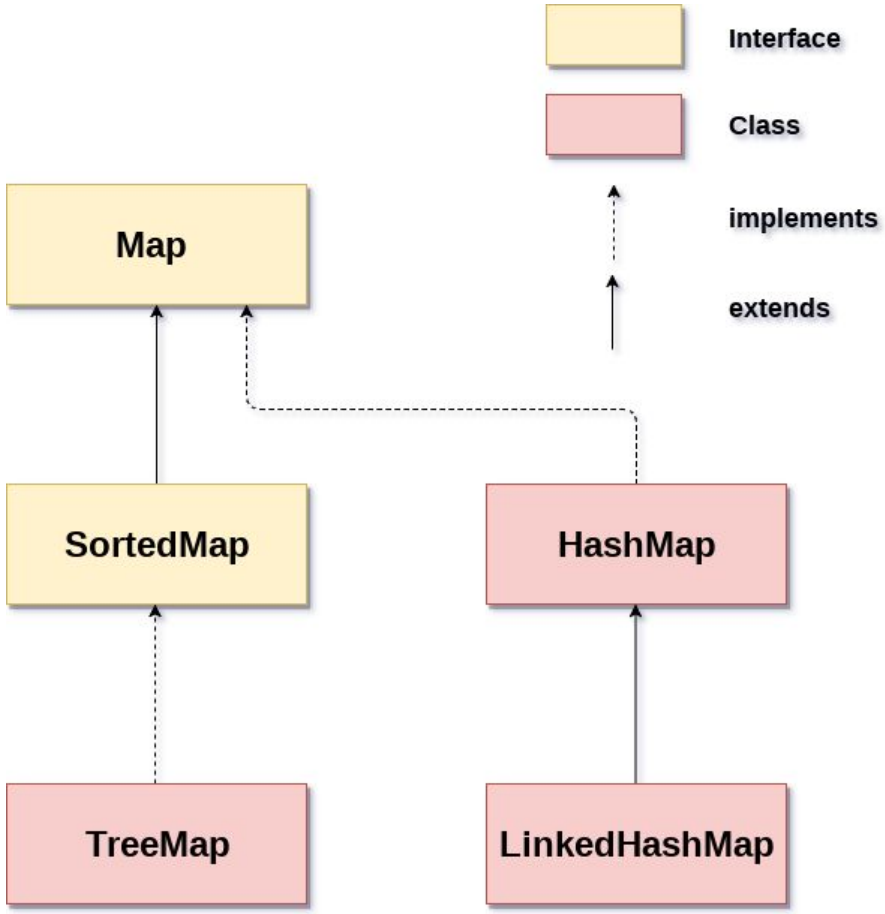
TreeSet: Esta clase implementa la interfaz SortedSet, que extiende la interfaz Set e impone un orden en los elementos. TreeSet utiliza un árbol para almacenar los elementos. Tienen el tiempo de recuperación más rápido de los Sets. Los elementos son ordenados y mantenidos de forma natural (numéricamente o alfabéticamente) y de manera ascendente. Los objetos deben implementar, si o si, el método compareTo().

Map

La interfaz **Map** no es una subinterfaz de Collection. Aunque ambas interfaces son parte del framework de colecciones de Java y se utilizan para almacenar elementos, hay algunas diferencias importantes entre ellas.

La interfaz Collection se utiliza para almacenar un conjunto de elementos individuales, mientras que la interfaz Map se utiliza para almacenar un conjunto de pares clave-valor. Cada elemento en un mapa se compone de una clave única y un valor asociado con esa clave. Los mapas se utilizan comúnmente para asociar valores con claves, y para realizar búsquedas eficientes de elementos basados en sus claves.

La interfaz Map define métodos para agregar, eliminar y buscar elementos en el mapa. Algunos de los métodos más comunes incluyen **put(key, value)** para agregar un elemento al mapa, **get(key)** para buscar un elemento en el mapa por su clave, y **remove(key)** para eliminar un elemento del mapa por su clave.


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

Implementaciones de Map

Al igual que las otras interfaces **Map** es implementada por distintas clases y otra interfaz:

HashMap: Esta clase utiliza una tabla hash para almacenar los pares clave-valor. Proporciona un acceso rápido y eficiente a los elementos del mapa, pero no los mantiene en un orden específico. Tampoco acepta valores nulos.

TreeMap: Esta clase implementa un árbol binario de búsqueda para almacenar los pares clave-valor. Los elementos se ordenan en función de sus claves, lo que significa que se puede iterar a través de ellos en orden ascendente o descendente.

LinkedHashMap: Esta clase también implementa una tabla hash para almacenar los pares clave-valor, pero mantiene un orden de inserción. Esto significa que los elementos se pueden iterar en el orden en que se agregaron al mapa.

Iterando un map

Una forma de iterar un mapa es utilizar Entry, que es una interfaz que representa una entrada (clave - valor) de un Map, para definir el tipo de dato del elemento que recorra el Map. Luego, en la parte de la derecha donde se escribe la colección, es llamado, por ésta, el método entrySet() que devuelve un Set de tipo Entry:

```
// Entry es una interfaz que representa una entrada (clave - valor) de un Map.
// Se utiliza el método entrySet() del Map que devuelve un Set de tipo Entry.
// El Set de tipo Entry es un conjunto de entradas de un Map.
// Entry tiene dos métodos getKey() y getValue() que nos permiten obtener la clave y el valor de la entrada.

for (Entry<Integer, String> alumno : linkedHashMap.entrySet()) {
    Integer clave = alumno.getKey();
    String valor = alumno.getValue();
    System.out.println(clave + "  ->  " + valor);
}
```

Diagrama de decisión para uso de colecciones Java